

c o n f e r e n c e

p r o c e e d i n g s

**The Second Symposium on
Operating Systems Design
and Implementation
(OSDI '96) Proceedings**

Seattle, Washington

October 28-31, 1996

Sponsored by

The USENIX Association

Co-sponsored by **ACM SIGOPS** and **IEEE TCOS**

USENIX[®]

The UNIX[®] and Advanced
Computing Systems Professional
and Technical Association



For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$20 for members and \$27 for nonmembers.
Outside the U.S.A. and Canada, please add
\$11 per copy for postage (via air printed matter).

Past OSDI Proceedings

OSDI '94 (First)	November, 1994	Monterey, California	\$20/27
------------------	----------------	----------------------	---------

1996 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-82-0

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

The USENIX Association

**Proceedings of the
Second Symposium on Operating Systems
Design and Implementation
(OSDI '96)**

Co-sponsored by ACM SIGOPS and IEEE TCOS

**October 28-31, 1996
Seattle Washington**

Program Co-Chairs

Karin Petersen, *Xerox PARC*
Willy Zwaenepoel, *Rice University*

Program Committee

Peter Chen, *University of Michigan*
Richard Draves, *Microsoft Research*
Carla Ellis, *Duke University*
Ed Felten, *Princeton University*
Jim Gray, *Microsoft Bay Area Laboratory*
Kevin Jeffay, *University of North Carolina*
David Johnson, *Carnegie Mellon University*
Jay Lepreau, *University of Utah*
Jeff Mogul, *Digital Equipment Corporation*
Marc Shapiro, *INRIA*
John Wilkes, *Hewlett-Packard Laboratories*
John Zahorjan, *University of Washington*

CONTENTS

Message from the Program Chairs	v
External Referees	vii
Author Index	viii

Tuesday, October 29

Opening Remarks and Awards Presentation

Karin Petersen, Xerox PARC; Willy Zwaenepoel, Rice University

Invited Talk: Java OS: Back to the Future	1
<i>Jim Mitchell, Sun Fellow, Vice President of Technology and Architectures, JavaSoft</i>	

Caching And Prefetching in I/O Systems

Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications	3
<i>Todd C. Mowry, Angela K. Demke and Orran Krieger, University of Toronto</i>	

A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching	19
<i>Tracy Kimbrel, University of Washington; Andrew Tomkins and R. Hugo Patterson, Carnegie Mellon University; Brian Bershad, University of Washington; Pei Cao, University of Wisconsin; Edward W. Felten, Princeton University; Garth Gibson, Carnegie Mellon University; Anna R. Karlin, University of Washington; and Kai Li, Princeton University</i>	

Efficient Cooperative Caching Using Hints	35
<i>Prasenjit Sarkar and John Hartman, University of Arizona</i>	

Distributed Shared Memory

Online Data-Race Detection via Choherency Guarantees	47
<i>Dejan Perkovic and Pete Keleher, University of Maryland</i>	

Lightweight Logging for Lazy Release Consistent Distributed Shared Memory	59
<i>Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves and Miguel Castro, IST -NESC</i>	

Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems	75
<i>Yuan Yuan Zhou, Liviu Iftode and Kai Li, Princeton University</i>	

Wednesday, October 30

Invited Talk: Active Networks	89
<i>David Tennenhouse, Telemedia, Networks and Systems Group, Laboratory for Computer Science, MIT</i>	

Scheduling and Synchronization

CPU Inheritance Scheduling	91
<i>Bryan Ford and Sai R. Susarla, University of Utah</i>	

A Hierarchical CPU Scheduler for Multimedia Operating Systems	107
<i>Pawan Goyal, Xingang Guo and Harrick M. Vin, University of Texas, Austin</i>	
The Synergy Between Non-blocking Synchronization and Operating System Structure	123
<i>Michael Greenwald and David Cheriton, Stanford University</i>	

OS Abstractions

Session Chair: Richard Draves, Microsoft Research

Microkernels Meet Recursive Virtual Machines	137
<i>Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, Steven Clawson, University of Utah</i>	

Making Paths Explicit in the Scout Operating System	153
<i>David Mosberger and Larry L. Peterson, University of Arizona</i>	

Performance Measurements

Studies of Windows NT Performance Using Dynamic Execution Traces	169
<i>Sharon E. Perl and Richard L. Sites, DEC SRC</i>	

Using Latency to Evaluate Interactive System Performance	185
<i>Yasuhiro Endo, Zheng Wang, J. Bradley Chen and Margo I. Seltzer, Harvard University</i>	

Thursday, October 31

Extensibility and Safety

Dynamic Binding for an Extensible System	201
<i>Przemyslaw Pardyak and Brian Bershad, University of Washington</i>	

Dealing With Disaster: Surviving Misbehaved Kernel Extensions	213
<i>Margo I. Seltzer, Yasuhiro Endo, Christopher Small and Keith A. Smith, Harvard University</i>	

Safe Kernel Extensions Without Run-Time Checking	229
<i>George C. Necula and Peter Lee, Carnegie Mellon University</i>	

Network Interfaces and Protocols

Session Chair: Dave Johnson, Carnegie Mellon University.

An Implementation of the Hamlyn Sender-Managed Interface Architecture	245
<i>Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich and John Wilkes, Hewlett-Packard Laboratories</i>	

Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems	261
<i>Peter Druschel and Gaurav Banga, Rice University</i>	

Effects of Buffering Semantics on I/O Performance	277
<i>Jose Carlos Brustoloni and Peter Steenkiste, Carnegie Mellon University</i>	

Message from the Program Chairs

Jay Lepreau successfully bore his baby two years ago, and, as he said himself, now it was up to us to raise OSDI well. Our first step in doing so was to select a strong program committee for OSDI '96. The researchers on the committee, eight from academia and six from industry, did a tremendous amount of work in an incredibly timely manner, each reading at least 34 of the 110 papers that were submitted. Our sincere thanks to all of them!

For this second symposium, the paper submission process differed from the previous one in two ways: we requested full paper submissions instead of abstracts to allow for more thoroughly presented submissions, and we extended the review period from six to twelve weeks to avoid the frenzy we endured last time.

On the other hand, we succeeded in maintaining the large number of, often lengthy, reviews for each of the submissions. The review process again consisted of two rounds. During the first round, we received an average 4.7 reviews for each paper. The results of this first round were then used to select fifty-two papers for the next round. In this second round, we asked for additional program committee and external reviews. In the end, papers in the second round got an average of 8.9 reviews, all of which were sent to the whole program committee before our meeting. We would like to thank all of our external reviewers for making this task possible and as painless as it was!

The program committee met for a full day at the Microsoft Bay Area Research Center in San Francisco. By then, most program committee members had digested a large number of the reviews, and each paper to be discussed had been read by at least five committee members. This led to often lively, but foremost informed, discussions. The result: 19 high quality papers were selected for publication in these proceedings, out of 110 submissions. Two papers clearly stood out and have therefore been selected as the best papers of the conference. The best paper awards go to:

Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications

Todd C. Mowry, Angela K. Demke, and Orran Krieger, *University of Toronto*

Safe Kernel Extensions Without Run-Time Checking

George C. Necula and Peter Lee, *Carnegie Mellon University*

Three of the accepted papers are co-authored by program committee members, out of eight such submissions. A significantly higher standard was applied to these submissions. Furthermore, as is customary by now, program committee authors did not see the reviews, nor participate in the discussions for their papers. Finally, all papers were shepherded by one program committee member before final publication. The shepherding process, in general, resulted in much improved versions of all papers, including those that were strong from the beginning.

This year, two invited talks complement the technical program. The talks focus on current technology changes, one coming from industry, *JavaOS*, and the other being proposed in academia, *Active Networks*. Both technologies seek to address the rapidly changing nature of devices connected to an ever-growing global network. It is our hope that these talks will foster fruitful discussions in new research areas for Operating Systems.

Of course, after putting together a technical program there is always the realization of things you could have done differently. We wish we had taken a stronger stand on accepting experience papers, and we wish we had accepted some of the less worked out, but more controversial idea-papers. However, we are happy with the result, and believe that the papers in this proceedings represent some of the best work being done in the field.

Many people were instrumental in helping us do our job. Early on, we could not have managed the incoming stream of submissions without the help of Ivy Jorgensen. Tim Miller helped with the software that handled submissions and notifications to the authors. On the committee, Jay Lepreau helped out with all his experience from running the show last time, and Jim Gray was generous enough to host the committee meeting and then take us out to sail on the San Francisco Bay. The USENIX staff was professional and supportive throughout. Ellie Young deserves special thanks for always quickly responding to every single one of our requests. Judy DesHarnais organized the conference; Toni

Veglia and Zanna Knight helped with the promotion; Pennfield Jensen dealt with the final publication; and finally, Dan Klein hung in there while we were pushing his tutorial deadlines as much as we possibly could. Many thanks to each of you!

Above all, we thank all the authors that submitted their work to continue making OSDI a world-class conference! The authors of accepted papers endured the committee's shepherding, and we truly appreciate your effort in producing the papers we believed would make this such a strong program.

Karin Petersen
Willy Zwaenepoel

October 1996

External Referees

The following people wrote often extensive reviews for one or more of the submitted papers, greatly assisting the papers' authors and the Program Committee.

Atul Adya	Mike Feeley	Kai Li	Prasenjit Sarkar
Thomas Anderson	Dror Feitelson	Wei Li	M. Satyanarayanan
Gregory R. Andrews	Wu-chi Feng	Jochen Liedtke	Stefan Savage
Godmar Back	Mary Fernandez	Bruce Lindsay	Wayne A. Sawdon
Jean-Loup Baer	Paulo Ferreira	Jim Lipkis	Michael L. Scott
Mary Baker	Bertil Folliot	Barbara Liskov	Bart Sears
Miche Baker-Harvey	Bryan Ford	Chris Maeda	Steven J. Sears
David E. Bakken	Martin Fouts	David A. Maltz	Margo Seltzer
Gaurav Banga	Michael Franklin	Mark Manasse	Liuba Shrira
Amnon Barak	Ron Frederick	Timothy Mann	Santosh Shrivastava
Brian Bershad	Syam Gadde	Bill Masek	Marcin Skubiszewski
Andrew P. Black	Benjamin Gamsa	David Mazieres	Donald R. Slutz
David L. Black	Gregory R. Ganger	Paul McJones	Chris Small
Matthias A. Blumrich	Kourosh Gharachorloo	Marshall K. McKusick	Mirjana Spasojevic
Hans-J. Boehm	Sanjay Ghemawat	Dylan McNamee	John F. Spaulding
Keith Bostic	Garth Gibson	Robert M. Melendez	Amitabh Srivastava
John Bridges	Richard Golding	Cliff Mercer	Jack Stankovic
Jose Brustoloni	H. Le Van Gong	Arif Merchant	Peter Steenkiste
Ramon Caceres	Ganesh Gopalakrishnan	Ethan Miller	Bjarne Steensgaard
Pei Cao	Allan Gottlieb	Frank Miller	David Steere
Mike Carey	Thomas Gross	Ron Minnich	Pat Stephenson
John Carter	Paulo Guedes	Greg Minshall	Ion Stoica
Jeff Chase	John H. Hartman	David Mosberger	Don Stone
Brad Chen	Carl Hauser	Eliot Moss	Jonathan Stone
David Cheriton	Maurice Herlihy	Sape J. Mullender	Michael Stumm
Frederic T. Chong	Mike Hibler	David Nagle	Mark Swanson
Alok Choudhary	Mark D. Hill	Greg Nelson	David Tennenhouse
Alex Colvin	Peter Honeyman	Rob Netzer	Douglas B. Terry
Alan Cox	Antony L. Hosking	Brian Noble	Marvin Theimer
Terence Critchlow	Jon Howell	Chris Nyberg	Chandramohan A. Thek-
Andrew S. Cromarty	Sugih Jamin	Douglas Orr	kath
Mark Crovella	Michael B. Jones	Jim O'Toole	Hideyuki Tokuda
Mike Dahlin	Linus Kamb	John Ousterhout	Jeff Turner
David DeWitt	Leonidas Kontothanassis	Susan Owicki	Robbert VanRenesse
Isabelle Demeure	Sacha Krakowiak	Venkata N. Padmanabhan	Mary K. Vernon
Peter Dibble	Orran Krieger	Przemyslaw Pardyak	Carl A. Waldspurger
Leendert van Doorn	Ravindra Kuramkote	Joseph Pasquale	Jonathan Walpole
Fred Douglass	Anthony LaMarca	R. Hugo Patterson	William E. Weihl
Peter Druschel	Carl Landwehr	Sharon Perl	Brent Welch
Dan Duchamp	Jim Larus	Larry Petersen	Walter Willinger
Kenneth Duda	Paul J. Leach	Guillaume Pierre	Ted Wobber
Derek Eager	Alvin R. Lebeck	Tom Pinckney	John Wroclawski
David A. Eckhardt	Tom Leblanc	James S. Plank	James C. Yee
E.N. Elnozahy	Edward K. Lee	Apratim Purakayastha	Michael Wayne Young
Thorsten von Eicken	Sam Leffler	Ragunathan Rajkumar	Matthew J. Zekauskas
Kevin Fall	Hank Levy	Mendel Rosenblum	Hui Zhang

Author Index

Godmar Back	137	Larry L. Peterson	153
Gaurav Banga	261	Prasenjit Sarkar	35
Brian Bershad	19, 201	Manuel Sequeira	59
Jose Carlos Brustoloni	277	Margo I. Seltzer	185, 213
Greg Buzzard	245	Richard L. Sites	169
Pei Cao	19	Cristopher Small	213
Miguel Castro	59	Keith A. Smith	213
J. Bradley Chen	185	Peter Steenkiste	277
David Cheriton	123	Sai R. Susarla	91
Steven Clawson	137	David Tennenhouse	89
Manuel Costa	59	Andrew Tomkins	19
Angela K. Demke	3	Patrick Tullman	137
Peter Druschel	261	Harrick M. Vin	107
Yasuhiro Endo	185, 213	Zheng Wang	185
Edward W. Felten	19	John Wilkes	245
Bryan Ford	91, 137	Yuanyuan Zhou	75
Garth Gibson	19		
Pawan Goyal	107		
Michael Greenwald	123		
Xingang Guo	107		
Paulo Guedes	59		
John Hartman	35		
Mike Hibler	137		
Liviu Iftode	75		
David Jacobson	245		
Anna R. Karlin	19		
Pete Keleher	47		
Tracy Kimbrel	19		
Orran Krieger	3		
Peter Lee	229		
Jay Lepreau	137		
Kai Li	19, 75		
Milon Mackey	245		
Scott Marovich	245		
Jim Mitchell	1		
David Mosberger	153		
Todd C. Mowry	3		
George C. Nacula	229		
Nuno Neves	59		
R. Hugo Patterson	19		
Przemyslaw Pardyak	201		
Dejan Perkovic	47		
Sharon E. Perl	169		

Invited Talk: JavaOS: Back To The Future

James G. Mitchell, *Sun Fellow, Vice President
Technology and Architecture, JavaSoft*

Abstract:

JavaOS provides just enough OS-like functionality to enable the Java environment to run on bare hardware.

JavaOS is not a modern operating system: it has no kernel, no virtual memory, and no diskfile system. On the other hand, it is aimed at the future: it is designed for a networked world. It removes almost all the hassles of owning and maintaining a personal computer. JavaOS has a graphical user interface, but no command line interpreter.

Written almost entirely in Java, it is small and fast.

Dr. Jim Mitchell has a Ph.D. from Carnegie-Mellon University and a B.S. from the University of Waterloo, Canada. He has worked on compilers, interactive programming systems, programming language design, document preparation systems, userinterface design, graphics hardware, distributed transactional filesystems, and distributed, object-oriented operating systems.

Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications

Todd C. Mowry, Angela K. Demke and Orran Krieger

*Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{tcm,demke,okrieg}@eecg.toronto.edu*

Abstract

Current operating systems offer poor performance when a numeric application's working set does not fit in main memory. As a result, programmers who wish to solve "out-of-core" problems efficiently are typically faced with the onerous task of rewriting an application to use explicit I/O operations (e.g., read/write). In this paper, we propose and evaluate a fully-automatic technique which liberates the programmer from this task, provides high performance, and requires only minimal changes to current operating systems. In our scheme, the compiler provides the crucial information on future access patterns without burdening the programmer, the operating system supports non-binding *prefetch* and *release* hints for managing I/O, and the operating system cooperates with a run-time layer to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. This approach maintains the abstraction of unlimited virtual memory for the programmer, gives the compiler the flexibility to aggressively move prefetches back ahead of references, and gives the operating system the flexibility to arbitrate between the competing resource demands of multiple applications. We have implemented our scheme using the SUIF compiler and the Hurricane operating system. Our experimental results demonstrate that our fully-automatic scheme effectively hides the I/O latency in out-of-core versions of the entire NAS Parallel benchmark suite, thus resulting in speedups of roughly twofold for five of the eight applications, with two applications speeding up by threefold or more.

1 Introduction

Many of the important computational challenges facing scientists and engineers today involve solving problems with very large data sets. For example, global climate modeling, computational physics and chemistry, and many engineering problems (e.g., aircraft simulation) can easily involve data sets that are too large to fit in main memory [7, 9, 23]. For such applications (which are commonly referred to as "out-of-core" applications), main memory simply constitutes an intermediate stage in the memory hierarchy, and the bulk of the data must reside on disk or other secondary storage. Ideally one could efficiently solve an out-of-core problem by simply taking the original in-core program and increasing the problem size. In theory, a paged virtual memory system could provide this functionality by transparently migrating data between main memory and disk whenever page faults occur. While this approach does yield a logically correct answer, the resulting performance is typically so poor that it is not considered a viable technique for solving out-of-core problems [35].

In practice, scientific programmers who wish to solve out-of-core problems typically write a separate version of the program with explicit I/O calls for the sake of achieving reasonable performance. Writing an out-of-core version of a program is a formidable task—it is not simply a matter of inserting a few I/O read or write statements, but often involves significant restructuring of the code, and in some cases can have a negative impact on the numerical stability of the algorithm [35]. Thus the burden of writing a second version of the program (and ensuring that it behaves correctly) presents a significant barrier to solving large scientific problems.

1.1 The Problem with Paged Virtual Memory

The performance of out-of-core applications that rely simply on paged virtual memory to perform their I/O is typically quite poor, as we will see later in Section 4. In our experiments, the performance loss is *not* due to limited I/O bandwidth (in fact, the disk utilization is fairly low), but rather to I/O *latency*, since each page fault causes the application to suffer the full latency of fetching the data from disk.

One can potentially achieve better performance by rewriting out-of-core applications to use explicit I/O calls (e.g., the read/write interface in UNIX) for the following three reasons. First, the non-blocking I/O calls provided by asynchronous I/O interfaces allow an application to hide latency by overlapping disk I/O with computation. For example, if non-blocking disk reads can be scheduled early enough, then all of the latency can potentially be hidden. In contrast, a disk read under paged virtual memory typically does not begin until it is triggered by a page fault, at which point the application suffers the full latency of the disk I/O. Second, explicit I/O calls can fetch a large number of blocks in a single request, which is important to fully exploit the underlying parallelism in high-bandwidth I/O systems (e.g., disk arrays). In contrast, page faults typically result in only a single outstanding page-sized read request at a time for a given process. (Although most operating systems attempt some form of page fault prefetching both to hide latency and to have multiple outstanding disk requests, it is difficult to do this efficiently for reasons we will discuss later in Section 2.2.) Finally, the explicit management of memory buffer space under explicit I/O allows the application to minimize memory consumption by immediately freeing (and if dirty, writing out to disk) any buffers containing data that will not be accessed again in the near future. Under paged I/O, since the memory manager lacks application-specific knowledge of future access patterns, it may make bad decisions and free pages that are about to be accessed. To avoid this problem, the memory manager is typically conservative by allocating more memory to the application than is actually required, which may result in poor resource utilization.

1.2 The Problem with Explicit I/O

While explicit I/O offers the potential for improved performance over paging, it unfortunately suffers from several disadvantages. The primary disadvan-

tage is the large burden placed on the programmer of rewriting an application to insert the I/O calls—our goal is to avoid this burden altogether. Another disadvantage is the performance overhead of these I/O system calls, which typically involve copying overhead to transfer data between the system's I/O buffers and the buffers managed by the application.

A third, less obvious disadvantage is that with explicit I/O, the application is implicitly making low-level policy decisions with its I/O requests (e.g., the size of the requests, and the amount of memory to be used for I/O buffering). However, the best policy decisions depend not only on application access patterns, but also on the physical resources available. Hence an application written assuming a particular amount of physical memory and disk bandwidth may perform poorly on a machine with a different set of resources, or in a multiprogrammed environment where some of the resources are being used by other applications. To illustrate how the available physical resources affect an application's performance, consider the amount of memory available for buffering I/O. If sufficient physical memory is available such that the entire data set can fit in memory, then an application with explicit I/O will pay the system call overhead with no benefit. On the other hand, if the application uses more buffer space for I/O than the available physical memory, then the buffers will suffer page faults, possibly resulting in worse performance than if the application had simply relied on paged virtual memory from the start.

1.3 Our Solution

To achieve high performance in out-of-core applications, we propose and evaluate a fully-automatic scheme for prefetching I/O whereby the operating system and the compiler cooperate to combine the advantages of both explicit I/O and paged virtual memory without suffering from the disadvantages. In our scheme, the compiler provides the crucial information on future access patterns without burdening the programmer, the operating system provides a simple interface for managing I/O which is optimized to the needs of the compiler, and a run-time layer accelerates performance by adapting to dynamic behavior and minimizing prefetch overhead. Our experimental results demonstrate that our scheme effectively hides the I/O latency in out-of-core versions of the entire NAS Parallel benchmark suite [2], thus resulting in speedups of roughly twofold for the majority of these applications, and over threefold in one case.

This paper is organized as follows. We begin in Section 2 by discussing how the compiler and the operating system can cooperate to automatically prefetch disk accesses for out-of-core applications. Next, in Sections 3 and 4, we describe our experimental framework and present our experimental results. Finally, in Sections 5, 6, and 7, we present related work, future work, and conclusions.

2 Automatically Tolerating I/O Latency

This section describes our system for automatically tolerating I/O latency. We begin by discussing the fundamental challenges that we have overcome, we then present an overview of our system, and finally we discuss the three major components of the system (i.e. the compiler, operating system, and runtime layer support) in more detail.

2.1 Fundamental Performance Issues

Our goal is to fully hide I/O latency, thus eliminating its impact on overall execution time. Conceptually, one can view our approach as enhancing the performance of virtual memory, since that is the abstraction we present to the programmer. Under paged virtual memory, an out-of-core application invokes two types of disk accesses: (i) faulting pages are read from disk into memory, and (ii) dirty pages are written out to disk to free up memory. Hiding write latency is reasonably straightforward since writes can be buffered and pipelined. Hiding read latency, on the other hand, is difficult because the application stalls waiting for the read (i.e. the page fault) to complete. The key to tolerating read latency is to split apart the *request* for data and the *use* of that data, while finding enough useful work to keep the application busy in between. We can accomplish this by *prefetching* pages sufficiently far in advance in the execution stream such that they reside in memory by the time they are needed.

Since prefetching does not reduce the number of disk accesses, but simply attempts to perform them over a shorter period of time, it cannot reduce the execution time of an application whose I/O bandwidth demands already outstrip the bandwidth provided by the hardware. Fortunately, we can construct cost-effective, high-bandwidth I/O systems by harnessing the aggregate bandwidth of multiple disks [5, 16, 28]. Roughly speaking, one can always increase the I/O bandwidth by purchasing

additional disks.¹

In addition to hiding I/O latency and providing sufficient I/O bandwidth, a third challenge in achieving high performance is effectively managing main memory, which can be viewed as a large, fully-associative cache of data that actually resides on disk. There are two issues here. First, to minimize page faults, we would like to choose the optimal page to evict from memory when we need to make room for new pages that are being faulted in. Toward this goal, most commercial operating systems use an approximation of LRU replacement to select victim pages. While LRU replacement may be a good choice for a default policy, there are cases where it performs quite poorly, and in such cases we would like to exploit application-specific knowledge to choose victim pages more effectively. The second issue is that we would like to minimize memory consumption, particularly when doing so does not degrade performance. For example, rather than filling up all of main memory with data that we are streaming through, we may be able to achieve the same performance by using only a small amount of memory as buffer space. By minimizing memory consumption, more physical memory will be available to the rest of the system, which is particularly important in a multiprogrammed environment. To accomplish both of these goals, we introduce an explicit *release* operation whereby the application provides a hint to the OS that a given page is not likely to be referenced again soon, and hence is a good candidate for replacement.

In summary, our approach overcomes the fundamental challenges of accelerating paged virtual memory as follows: (i) *prefetches* are used to tolerate disk read latency, (ii) *multiple disks* are used to provide high-bandwidth I/O, and (iii) *release* operations are used to effectively manage memory. We now discuss the overall structure of our software system.

2.2 Software Architecture Overview

To prefetch and release data effectively, we need detailed knowledge of an application's future access patterns. Although one might attempt to deduce this information from inside the OS by looking for repeated patterns in the access history, such an approach would be limited only to simple access patterns (e.g., even the simple indirect refer-

¹There are more subtle issues involved with increasing I/O bandwidth, of course. However, this approach does appear to be promising enough for our purposes, and exploring I/O bandwidth issues further is beyond the scope of this paper.

ences that commonly occur in sparse-matrix applications would be extremely difficult for the OS to predict), and would require adding additional complexity to the OS, which is something we wish to avoid.² Instead, we turn to the compiler to provide information on future access patterns, since it has the luxury of being able to examine the entire program all at once. Also, by using the compiler to extract this information automatically, we avoid placing any burden on the programmer, who continues to enjoy the abstraction of unlimited virtual memory.

2.2.1 The Compiler / OS Interface

Given that the compiler will be extracting and passing access pattern information to the OS, an important question is what form this interface should take. Note that this interface will *only* be used by the compiler, and *not* by the programmer—the programmer's interface will be unlimited virtual memory, and the compiler and operating system cooperate to preserve this illusion. Ideally, we would like an interface that requires minimal complexity within the OS (so that it can be readily incorporated into an existing commercial OS), and which maximizes the compiler's ability to improve performance, given the strengths and weaknesses of realistic compilation technology.

One possibility would be for the compiler to pass a summary of future access patterns to the OS through a single call at the start of execution. However, from the compiler's perspective, this approach is undesirable since the access patterns in real applications often depend on dynamic control and data dependencies that can only be resolved at run-time. For example, in the bucket sort application (BUK) discussed later in this paper, the important data accesses are indirect references based on the contents of a large array. The values in this array are unknown at startup time; but even if they were known, passing this very large array along with a description of how to use it to compute addresses would greatly complicate not only the interface and the compiler, but also the OS, which would ultimately be responsible for generating the addresses. Another disadvantage of this approach is that it pushes the complexity of matching up the access patterns with *when* those accesses actually take place into the OS. For example, if the compiler indicates that the program will be streaming through a large array, it is not helpful if the OS brings the data into

memory too fast (or too slow) relative to the rate at which it is being consumed. Since tracking an application's access patterns means that the OS must see either page faults or explicit I/O on a regular basis, it is unclear that this interface offers any less overhead than an interface requiring regular system calls. Hence we will focus instead on an interface where prefetch addresses are passed in at roughly the time when the prefetch should be sent to disk, and where release addresses are passed in when the data is no longer needed.

The next logical question is whether we can simply compile to an existing asynchronous read/write I/O interface, or whether a new interface is actually needed. There are two reasons why existing read/write I/O interfaces are unacceptable for our purposes. First, for the compiler to successfully move prefetches back far enough to hide the large latency of I/O, it is essential that prefetches be *non-binding* [19]. The non-binding property means that when a given reference is prefetched, the data value seen by that reference is bound at *reference* time; in contrast, with a binding prefetch, the value is bound at *prefetch* time. The problem with a binding prefetch is that if another store to the same location occurs during the interval between a prefetch and a corresponding load, the value seen by the load will be stale. Hence we cannot move a binding prefetch back beyond a store unless we are certain that they are to different addresses—unfortunately, this is one of the most difficult problems for the compiler to resolve in practice (i.e. the problem of “alias analysis”, also known as “memory disambiguation” or “dependence analysis”). Since an asynchronous I/O read call implicitly renames data by copying it into a buffer, it is a binding prefetch. To illustrate this problem, consider the code in Figure 1(a). If we use the read/write interface, we might generate code similar to Figure 1(b). Unfortunately, this code produces an incorrect result if the parameters *a* and *b* are aliased (e.g., `foo(&X[0], &X[0])`) or even partially overlap (e.g., `foo(&X[10], &X[0])`). To implement non-binding prefetching, the data should have the same name (or address) both in memory and on disk, which corresponds to the abstraction of paged virtual memory. Figure 1(c) shows the preferred code which uses non-binding prefetch and release operations, and always produces a correct result.

The second problem with an asynchronous read/write interface is that it compels the OS to perform an I/O access. Instead, we would prefer to give the OS the flexibility to drop requests if doing so might achieve better performance, given

²This additional complexity may increase the critical page fault path in the OS, and hence degrade application performance.

<pre>foo(double *a, double *b) { /* Assume that a & b reside /* on disk at this point. */ ... for (i = 0; i < 100; i++) { a[i+1] = a[i] + b[i]; } }</pre>	<pre>foo(double *a, double *b) { double a_buf[101], b_buf[100]; /* Read a & b from disk into buffers. */ read(a, &a_buf[0], 101*sizeof(double)); read(b, &b_buf[0], 100*sizeof(double)); ... for (i = 0; i < 100; i++) { a_buf[i+1] = a_buf[i] + b_buf[i]; } /* Write a_buf back out to disk. */ write(a, &a_buf[0], 101*sizeof(double)); }</pre>	<pre>foo(double *a, double *b) { /* Prefetch a & b into memory. */ prefetch(a, 101*sizeof(double)); prefetch(b, 100*sizeof(double)); ... for (i = 0; i < 100; i++) { a[i+1] = a[i] + b[i]; } /* Finished with a & b. */ release(a, 101*sizeof(double)); release(b, 100*sizeof(double)); }</pre>
(a) Original Code	(b) Read/Write Interface	(c) Prefetch/Release Interface

Figure 1: Example illustrating the importance of non-binding prefetches.

the dynamic demands for and availability of physical resources. For example, if there is not enough physical memory to buffer prefetched data, or if the disk subsystem is overloaded, we may want to drop prefetches. Hence the preferred interface is a natural extension of paged virtual memory which includes *prefetch* and *release* as non-binding performance hints, thus giving the compiler the flexibility to aggressively move prefetches back ahead of references, and giving the OS the flexibility to arbitrate between the competing resource demands of multiple applications. (Note that the “MADV_WILLNEED” and “MADV_DONTNEED” hints to the `madvise()` interface can potentially be used to implement prefetch and release in UNIX.)

2.2.2 Minimizing Prefetch Overhead

Earlier studies on compiler-based prefetching to hide cache-to-memory latency have demonstrated the importance of avoiding the overhead of unnecessarily prefetching data that already resides in the cache [19, 20]. To address this problem, compiler algorithms have been developed for inserting prefetches only for those references that are likely to suffer misses. An analogous situation exists with I/O prefetching, since we do not want to prefetch data that already resides in main memory—hence, we perform similar analysis in our compiler (as we discuss later in Section 2.3). Unfortunately, it is considerably more difficult to avoid unnecessary prefetches with I/O prefetching since main memory is so much larger than a cache that our loop-level compiler analysis tends to underestimate its ability to retain data. As a result, unnecessary prefetches do occur, and we must be careful to minimize their overhead.

Compared with cache-to-memory prefetching, where the overhead of an unnecessary prefetch is

simply a wasted instruction or two³, the overhead of an unnecessary I/O prefetch is considerably larger since it involves making a system call and checking the page table before discovering that the prefetch can be dropped. To reduce this overhead, we introduce a run-time layer in our system which keeps track at the *user level* of whether pages are believed to be in memory or not. Therefore we can typically drop unnecessary prefetches immediately without performing a system call, and we have found this to be essential in achieving high performance.

Having introduced the three layers of our system—the compiler, the OS, and the run-time layer—we now discuss each layer in more detail.

2.3 Compiler Support

The bulk of our compiler algorithm is a straightforward extension of an algorithm that was developed earlier for prefetching cache-to-memory misses in dense-matrix and sparse-matrix codes [19, 20]. Roughly speaking, we changed the input parameters that describe the cache size, line size, and miss latency to correspond to main memory size, the page size, and the page fault latency, respectively. Based on this memory model, the compiler uses *locality analysis* to predict when misses (i.e. page faults) are likely to occur, it isolates these faulting instances through *loop splitting* techniques, and schedules prefetches early enough using *software pipelining*. Figure 2 shows an example of the output of our compiler for a simple loop body (notice that it is able to prefetch the indirect `a[b[i]]` reference as well as the dense `b[i]` and `c[i][j]`

³Unnecessary cache prefetches are dropped as soon as the primary cache tags are checked. The overhead is simply the prefetch instruction, plus any instructions needed to generate the prefetch address, plus one cycle of wasted cache tag bandwidth.

<pre> int a[1000000]; int b[1000000]; int c[1000000][8]; for (i = 0; i < 1000000; i++) for (j = 0; j < 8; j++) a[b[i]] = a[b[i]] + c[i][j]; </pre>	<pre> prefetch_block(&b[0], 8); prefetch_block(&c[0][0], 4); for (i = 0; i < 128; i++) prefetch(&a[b[i]]); /* Note: 995328 = ($\lfloor \frac{1000000}{4096} \rfloor - 1$) * 4096 */ for (i1 = 0; i1 < 995328; i1 += 4096) { prefetch_release_block(&b[i1+8192], &b[i1-1], 4); for (i0 = i1; i0 < i1 + 4096; i0 += 512) { prefetch_release_block(&c[i0+512][0], &c[i0-1][0], 4); for (i = i0; i < i0 + 512; i++) { prefetch(&a[b[128+i]]); for (j = 0; j < 8; j++) a[b[i]] = a[b[i]] + c[i][j]; } } } for (i = 995328; i < 1000000; i++) for (j = 0; j < 8; j++) a[b[i]] = a[b[i]] + c[i][j]; </pre>
(a) Original Code	(b) Code with Prefetching

Figure 2: Example of the output of the prefetching compiler. (The first argument to all prefetch calls is the prefetch address; the second argument to `prefetch_release_block` is the release address; the final argument to “block” versions is the number of 4KB pages to be fetched and/or released.)

references). Since space limitations prevent us from describing the compiler algorithm in detail, we focus mainly on the major changes to the original algorithm [19].

Two of our modifications to support I/O prefetching are related to spatial locality—i.e. when strided accesses fall within the same page—in which case page faults only occur on iterations that cross page boundaries. First, we use *strip mining* [24] rather than *loop unrolling* to isolate these faulting iterations, since replicating a loop body 1000 times or more is clearly infeasible. Notice in Figure 2(b) that loop *i* has been strip mined twice (into loops *i0* and *i1*) to account for the spatial locality of `b[i]` and `c[i][j]`. (The *i* loop has been strip mined twice since `c[i][j]` accesses data more quickly than `b[i]`, and therefore needs to be prefetched at a faster rate.) Second, to fully exploit the available bandwidth in our I/O subsystem, we prefetch several pages at a time for references with spatial locality (e.g., four pages are fetched at a time for `b[i]` and `c[i][j]`⁴). (Note that for references without spatial locality—e.g., `a[b[i]]`—we prefetch only a single page at a time.) Similarly, we convert the prolog loops from the original algorithm into block prefetches whenever possible, as shown in the first two lines of Figure 2(b).

Generating release operations is straightfor-

ward. The compiler already identifies groups of references that effectively share the same data and can be treated as a single reference—this is called “group locality”. For each of these groups (a group may potentially contain only a single reference), the compiler identifies the *leading reference* (i.e. the first reference to access the data) as the reference to prefetch—we simply extend this analysis to also identify the *trailing reference* (the last one to touch the data) as the address to release. (Note that for indirect references such as `a[b[i]]`, we do not generate a release operation since it is too difficult to predict whether the data will be accessed again soon.) To minimize system call overhead, we bundle prefetch and release requests together whenever appropriate, as illustrated in Figure 2(b).

Perhaps the most significant change we made to the original algorithm is to reason more carefully about loop bounds and array bounds that are small relative to a page size. This was less of a concern for cache-to-memory prefetching due to the relatively small size of cache lines. However, it is common to find inner loops (and sometimes even surrounding loops) which access less data than a 4 KB page (e.g., the *j* loop in Figure 2(a)). Attempting to software pipeline prefetches across such loops is ineffective, since the pipeline never gets started. Instead, our compiler pipelines the prefetches across the first surrounding loop which touches more than a page of the given array, as illustrated by the fact that prefetches for `c[i][j]` are pipelined along the

⁴The number of pages to fetch in a block is a parameter which can be specified to the compiler. We chose four arbitrarily for this example.

i loop rather than the j loop in Figure 2(b). Having described the compiler support for I/O prefetching, we now focus on the other half of the equation: the operating system.

2.4 Operating System and Run-Time Layer Interaction

To support compiler directed prefetching, the OS needs to be able to respond to the prefetch and release operations issued by the application. This functionality is easy to add since the OS is already able to unmap pages of memory and initiate asynchronous requests to the file system. One issue is how the OS should handle prefetch requests when there is no free memory available. Since we expect the application/compiler to be managing memory requirements, the OS simply drops prefetches when all memory is in use.

To help the run-time layer reduce the overhead of prefetching, the OS also provides applications with a single physical memory page that is shared with the OS. Applications that prefetch are required to register with the OS to initiate sharing. The shared page is used as a bit vector with each bit representing one or more contiguous pages of the application's virtual memory space (a set bit indicates that the corresponding page is in memory). The granularity of the bit vector is determined by the run-time layer at program start-up. Bits are set by the run-time layer when a prefetch request is issued, and by the OS when non-prefetched page faults occur. The OS also clears bits when release requests are issued and when the memory manager reclaims pages.

The run-time layer uses the bit vector to *filter* the prefetches inserted by the compiler by checking to see if the requested page is already in memory. In many cases this simple test can avoid the cost of a system call to the OS, thus reducing overhead. For block prefetch requests, we check each page until one is found that is not in memory, then pass all remaining pages to the OS. In this way, at most one system call is required for a block prefetch.

3 Experimental Framework

We now describe our experimental platform, and the applications which we study in our experiments.

3.1 Experimental Platform

The experimental platform used to evaluate our scheme is the Hurricane file system [16] and Hurri-

Table 1: Experimental platform characteristics.

Processor	
Processor type:	Motorola 88100
Clock rate:	16.67 MHz
Data cache size:	16KB
Instruction cache size:	16KB
Physical Memory	
Total size:	64 MBytes
Available to application:	48 MB
Disks	
Number of disks:	7
Maximum transfer rate:	640 KB/sec
Average rotational latency:	8.61 msec
Track-to-track seek time:	5 msec
Kernel Operation Overhead	
IPC request:	70 μ sec
In-core fault:	200 μ sec
Out-of-core fault:	800 μ sec
Base prefetch:	60 μ sec
+ per out-of-core page:	200 μ sec
+ per in-core page:	30 μ sec
+ per in-page table page:	10 μ sec
File System Operation Overhead	
Prefetch (per-page):	70 μ sec
Read/Write (per-page):	70 μ sec

cane operating system [33] running on the Hector shared-memory multiprocessor [34]. Hurricane is a hierarchically clustered, micro-kernel based operating system that is mostly POSIX compliant. It was largely irrelevant that the system was a multiprocessor; we chose this platform because the system has multiple disks attached to it, the file system can stripe a single file across multiple disks, and the operating system could be modified to add prefetch and release operations. For all experiments shown in subsequent sections, the pages of the applications are striped by the file system round-robin across all seven disks. An extent-based policy is used to store the file on each of the disks, where contiguous file blocks are stored to contiguous blocks on the disk to avoid seek operations for sequential file accesses. The disk scheduler treats prefetches the same as normal disk read requests.

In addition to adding prefetch and release operations to Hurricane, we also added extensive instrumentation to enable us to produce the detailed statistics shown in subsequent sections. The basic characteristics of our experimental platform (with the instrumentation disabled) are shown in Table 1, and more detailed descriptions of the platform can be found in earlier publications [16, 33, 34].

We believe that our experimental results are conservative for the following reasons: (i) instrumentation is enabled for all the experiments, and

Table 2: Description of applications.

Name	Description	Input Data Set	Memory Required		Original Execution Time (mins)
			Absolute	% of Available	
BUK	integer bucket sort algorithm	2^{23} 19-bit integers	103 MB	215 %	21.0
CGM	solves an unstructured sparse linear system using the conjugate gradient method	28000x28000 sparse matrix with 7,607,024 non-zeros	103 MB	215%	57.2
EMBAR	monte-carlo simulation	2^{24} random numbers	134 MB	279%	53.9
FFTPDE	3-D FFT PDE, performs forward and inverse FFT's	128x128x128 matrix of complex numbers	117 MB	244%	87.9
MGRID	computes 3-D scalar potential field on a uniform cubical grid using a multigrid solver	128x128x128 matrix	58 MB	121%	31.9
APPLU	solves four coupled parabolic / elliptic PDE's using SSOR method to invert jacobian matrix	5x5x64x64x32 matrices	120 MB	250%	48.9
APPSP	solves five coupled parabolic / elliptic PDE's using diagonalized approximate factorization method	90x90x90 matrices	117 MB	244%	224.3
APPBT	solves three coupled parabolic / elliptic PDE's using block approximate factorization method	5x5x64x64x32 matrices	94 MB	196%	85.2

hence the system overheads are significantly inflated; (ii) the operating system overhead is also inflated because the hardware does not support cache coherence, and hence many of the operating system data structures are accessed in an uncached state; and (iii) processor speeds have increased more rapidly than disk speeds, and hence the importance of tolerating I/O latency has increased in modern systems.

3.2 Applications

To evaluate the effectiveness of our approach, we measured its impact on the performance of the entire NAS Parallel benchmark suite [2]. We chose these applications because they represent a variety of different scientific workloads, their data sets can easily be scaled up to out-of-core sizes, and they have not been written to manage I/O explicitly. Our goal is to show that these scientific benchmarks can achieve high performance with out-of-core data sets without requiring any extra effort to rewrite the program. Because these programs were originally written to evaluate processor performance, they all generate a data set at start-up, perform a series of computations, and then discard the results. To make the programs more realistic, we modified them to use a pre-initialized data set and write their results back out to disk. This was achieved by using a mapped file I/O interface—the data accesses have not been modified but the data now comes from disk. An exception to this strategy is EMBAR where a random initialization is performed once for every iteration and separation would not

be appropriate. A brief description of each of the benchmarks and the data set used is given in Table 2.

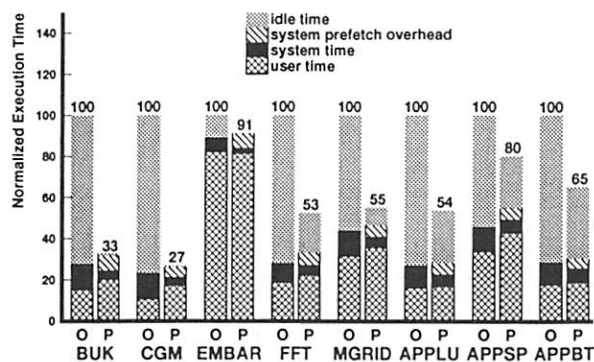
We implemented our prefetching algorithm as a pass in the SUIF (Stanford University Intermediate Format) compiler [31], which we used to convert the original Fortran source code of each application into C code containing prefetch and release calls (as illustrated earlier in Figure 2(b)). We then compile this resulting C code into a Hurricane executable using gcc version 2.5.8 with the -O2 optimization flag.

4 Experimental Results

We now present the results of our experiments. We begin by focusing on the impact of our scheme on overall execution time, including the effectiveness of the compiler and the run-time layer. We then look at the performance from a system-level perspective, including the effects on disk and memory utilization.

4.1 Performance Improvement

Figure 3(a) shows the overall performance improvement achieved through our automatic prefetching scheme. For each application, we show two bars representing normalized execution time: the original program relying simply on paged virtual memory to perform its I/O (O), and the program once it is compiled to use prefetching (P). In each bar, the top section is the amount of time when the proces-



(a) Overall Performance
(O = original, P = with prefetch)

Benchmark	Original		With Prefetch		
	Total Faults (x1000)	Avg. Stall Time (msec)	Total Faults (x1000)	Avg. Stall Time (msec)	I/O Stall Reduction (%)
BUK	41.529	24.5	0.810	16.1	98.7%
CGM	135.066	22.0	0.207	26.5	99.8%
EMBAR	65.535	7.7	0.005	13.5	100.0%
FFTPDE	135.646	31.1	28.432	39.3	73.6%
MGRID	62.231	19.9	7.642	24.2	85.1%
APPLU	91.220	26.3	31.663	26.4	65.2%
APPSP	412.234	20.5	143.996	26.2	55.4%
APPBT	156.172	26.2	77.035	25.6	51.9%

(b) I/O Stall Statistics

Figure 3: Overall performance improvement from prefetching

sor was idle, which corresponds roughly to the I/O stall time since we run only a single application during these experiments. The bottom section of each bar is the time spent executing in user mode—for the prefetching experiments, this includes the instruction overhead of issuing prefetches, including any overhead in the run-time layer of checking the bit vector to filter out unnecessary prefetches. The middle sections of each bar are the time spent executing in system mode. For the original programs, this is the time required for the operating system to handle page faults; for the prefetching programs, we also distinguish the time spent in the OS performing prefetch operations.

As we see in Figure 3(a), the speedup in overall performance ranges from 9% to 270%, with the majority of applications speeding up by more than 80%. Figure 3(b) presents additional information on page faults⁵ and stall time. As we see in Figure 3(b), more than half of the I/O stall time has been eliminated in seven of the eight applications,

⁵Throughout this discussion, we will refer to page faults that cause the application to stall waiting for I/O simply as faults, and ignore page faults for in-core data.

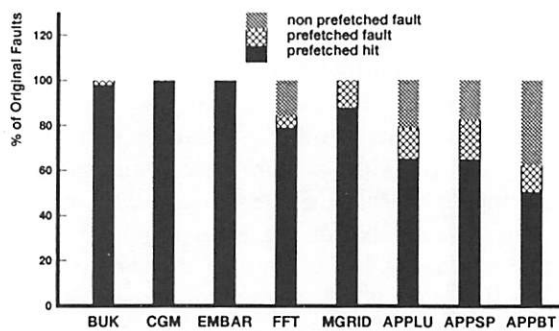
with three applications eliminating over 98% of their I/O stall time.

Having established the benefits of our scheme, we now focus on the costs. Figure 3(a) shows that the instruction overhead of generating prefetch addresses and checking whether they are necessary in the run-time layer causes less than a 20% increase in user time in five of the eight applications—in the worst case (CGM), the user time increases by 70%. However, in all cases this increase is quite small relative to the reduction in I/O stall time. If we focus on the system-level overhead of performing prefetch operations, we see in Figure 3(a) that in most cases this overhead is directly offset by a reduction in system-level overhead for processing page faults. Hence the overheads of our scheme are low enough to translate into significant overall performance improvements in all of these applications.

We wish to emphasize that all of these results are fully automatic—we have not rewritten any of the applications or modified the code generated by the compiler. Having discussed the performance at a high level, we now focus on the compiler and run-time layer in more detail.

4.1.1 Effectiveness of the Compiler and Run-Time Layer

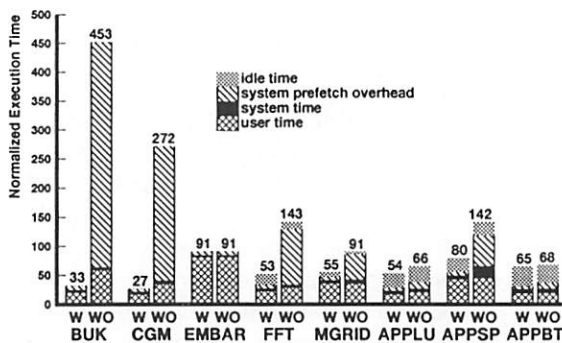
Figure 4 presents additional information which is useful for evaluating how effective our compiler is at inserting prefetches appropriately, and how effective the run-time layer is at minimizing prefetching overhead. Figure 4(a) shows a breakdown of the impact of prefetching on the original page faults in the application. This breakdown contains three categories: (i) those that were prefetched and successfully eliminated page faults (*prefetched hit*), (ii) those that were prefetched but remained page faults (*prefetched fault*), and (iii) those that were not prefetched (*non-prefetched fault*). The combination of the first two cases is often referred to as the *coverage factor* (i.e. the fraction of original page faults that were prefetched). For all cases except APPBT, the coverage factor is greater than 75% (in four cases, it is greater than 99%). Most of the page faults that we failed to prefetch were due to inner loops with small loop bounds, where the fact that the bound was small could not be determined at compile time. For example, if the *j* loop in example in Figure 2(a) had an upper bound of *N* which turned out to be small at run-time (but which we could not determine at compile time), and if the dimensions of the *c* matrix were also unknown at compile time, our compiler can make the mistake of software pipelining references across the



(a) Coverage Factor

Benchmark	Unnecessary Prefetches Issued to OS	Inserted Prefetches Filtered at Run-Time
BUK	0.07%	99.79%
CGM	0.08%	99.74%
EMBAR	0.00%	0.02%
FFTPDE	7.99%	99.59%
MGRID	8.03%	99.17%
APPLU	3.75%	96.99%
APPSP	7.55%	99.51%
APPBT	2.54%	98.31%

(b) Unnecessary Prefetches



(c) Performance of prefetching with (W) and without (WO) filtering (normalized to the original, non-prefetched case).

Figure 4: Effectiveness of the compiler analysis and run-time filtering.

j loop rather than the i loop. Situations like this can cause us to miss important prefetches, since the software pipeline never gets started. This problem can be fixed through a straightforward extension of our compiler algorithm whereby we create two versions of the loop, and choose the proper one to execute by testing the loop bound at run-time.

The effectiveness of our compiler in scheduling prefetches the right amount of time in advance is reflected by the size of the *prefetched fault* cate-

gory in Figure 4(a). A large value means that the prefetches are either not issued early enough, in which case the page has not arrived in memory by the time it is referenced, or are issued too early, in which case the page has already been flushed from memory before it is referenced. In the cases where this category is noticeable in Figure 4(a), the problem is almost always that the prefetches were not issued early enough. However, given how large I/O latency is, it is encouraging that this case is generally small relative to the number of successful prefetches.

To evaluate the effectiveness of the run-time layer at reducing prefetching overhead, Figure 4(b) presents statistics on how many prefetches were *unnecessary* (i.e. the page was already mapped into memory). (Note that a prefetch for a page that is in memory but is on the free list is not considered to be unnecessary, since it performs useful work by reclaiming the page.) The left-hand column of Figure 4(b) shows that almost all of the prefetches issued to the system by the run-time layer are useful. All unnecessary prefetches that are issued to the system occur as part of a block prefetch request in which prefetching is required for at least one page. The right-hand column of Figure 4(b) shows the fraction of dynamic prefetches that were inserted by the compiler which turn out to be unnecessary, and are filtered out by the run-time layer. For reasons discussed earlier in Section 2.2.2, it is difficult for our compiler to avoid inserting unnecessary prefetches, and we see that over 96% of the prefetches were unnecessary for all but EMBAR (where the access patterns are simple enough that the compiler's analysis is perfect).

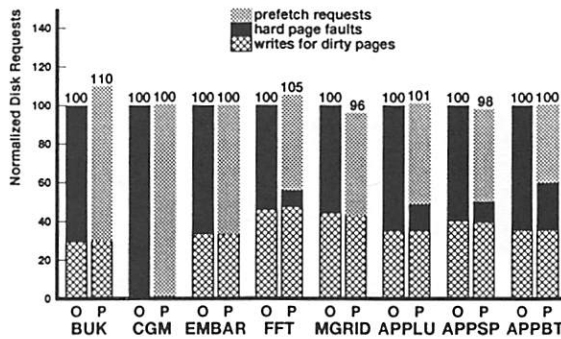
Figure 4(c) quantifies the performance advantage of the run-time layer. As we see in Figure 4(c), half of the applications (BUK, CGM, FFT and APPSP) run slower than the original non-prefetching versions when the run-time layer is removed. This is not surprising since the overhead of dropping an unnecessary prefetch in the run-time layer is roughly 1% as expensive as issuing it to the OS. Hence the run-time layer is clearly essential.

4.2 Disk and Memory Utilization

In Figure 5 we break down the types of requests seen by the disks and show average disk utilization during execution for both the original and prefetching versions of the applications. In almost all cases, the total disk requests do not increase as a result of prefetching, and for two of the applications they actually decrease as prefetches prevent the system

Table 3: Memory sub-system activity and amount of free memory

Benchmark	Original			With Prefetch and Release			
	Pages Freed by System (pages)	Minimum Free Memory (%)	Average Free Memory (%)	Pages Freed by System (pages)	Pages Freed by release (pages)	Minimum Free Memory (%)	Average Free Memory (%)
BUK	68916	5.8%	26.9%	3461	41729	29.2%	73.7%
CGM	125817	14.8%	21.1%	125710	834	7.3%	23.4%
EMBAR	55647	15.0%	22.0%	0	65504	98.4%	98.5%
FFTPDE	146699	14.9%	20.9%	156463	7164	9.9%	26.0%
MGRID	59181	14.3%	23.4%	60349	0	12.3%	25.9%
APPLU	82978	11.9%	25.0%	84395	0	7.9%	28.9%
APPS	450507	10.5%	18.6%	448732	17196	9.0%	35.4%
APPBT	148174	11.3%	22.8%	148580	516	11.7%	25.5%



(a) Disk activity

Benchmark	Original	With Prefetch
BUK	11.8%	40.1%
CGM	11.6%	46.0%
EMBAR	5.9%	9.0%
FFTPDE	18.9%	35.1%
MGRID	15.8%	29.0%
APPLU	18.6%	31.8%
APPS	16.3%	20.7%
APPBT	15.8%	20.1%

(b) Average disk utilization

Figure 5: Breakdown of requests sent to disk and average utilization (O = original program, P = with prefetch)

from writing out dirty pages that will be referenced again soon. Hence the increased disk utilization shown in Figure 5(b) is simply due to the fact that we are performing roughly the same number of disk accesses over a shorter period of time.

Finally, Table 3 summarizes memory usage during each application's execution. Since our current compiler implementation is not aggressive about inserting release operations, most applications do not contain a significant number of them. However, when release operations are used (e.g., BUK and EMBAR), we see that a large percentage of mem-

ory is kept free at all times since only the portion of the data set actually being used is kept in memory. We expect that this would greatly reduce the impact of an out-of-core program on other applications in a multiprogrammed environment, and we intend to explore this issue further in future work.

4.3 Problem Size Variations

Having demonstrated the benefits of I/O prefetching where the problem size is roughly twice as large as the available memory, we now look at the performance when the problem size is varied.

4.3.1 In-Core Problem Sizes

We begin with cases where the data sets fit within main memory. In these cases, we would expect prefetching to degrade performance, since the prefetches incur overhead but provide little or no benefit. Figure 6 shows two sets of experiments—the cold-started and warm-started cases—on data sets that are roughly 10-35% as large as the available memory. Starting with the cold-started cases, we see that prefetching degrades performance in four cases, but actually *improves* performance in three cases (BUK, APPLU, and APPBT) by hiding the latency of cold page faults. To further isolate the prefetching overhead, we also warm-started the applications by preloading all of their data from the input files into memory before timing the runs. As expected, prefetching typically degrades performance in the warm-started cases since it offers no potential advantage. However, we believe that the cold-started cases are more realistic for most applications, since real programs must read their input data from disk.

In these experiments, we made no attempt to minimize prefetching overhead for in-core data sets, but this is a problem that we are planning to ad-

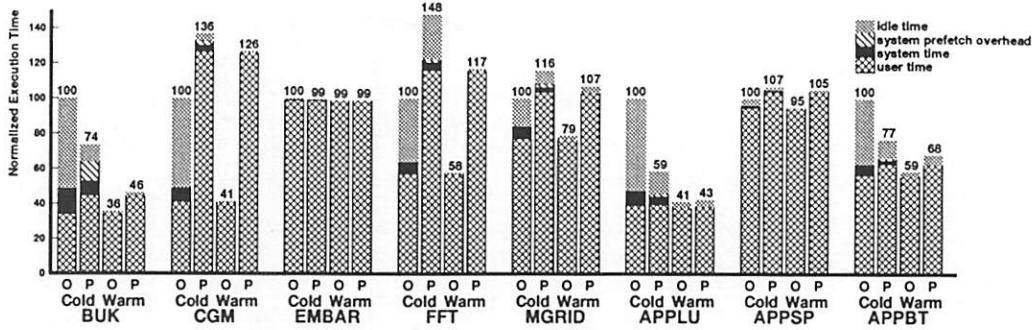


Figure 6: Performance with in-core data sets (O = original, P = with prefetch; Cold = cold-started, Warm = warm-started). Performance is normalized to the original, cold-started cases.

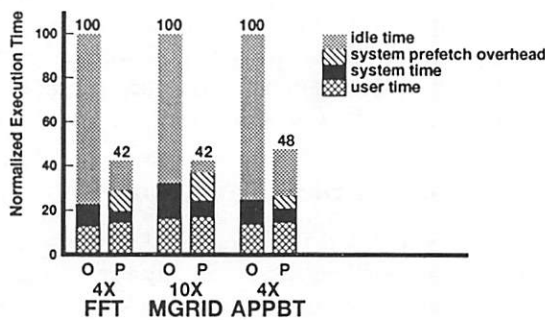


Figure 7: Performance with larger out-of-core problem sizes. Numbers above application names indicate how much larger the problem sizes are than available memory.

dress in future work. In particular, we can generate code that dynamically adapts its behavior by comparing its problem size with the available memory at run-time, and suppressing prefetches (after the cold faults have been prefetched in) if the data fits within memory. The fact that I/O prefetching can still potentially improve performance even on relatively small data sets by hiding cold page faults is an encouraging result.

4.3.2 Larger Out-of-Core Problem Sizes

In addition to looking at smaller problem sizes, we also experimented with much larger data sets than our earlier out-of-core problem sizes. Figure 7 shows the performance of three applications where the problem size is 4-10 times larger than the available memory. Recall that for MGRID, our earlier problem size was only 20% larger than the available memory—the next larger problem size (shown in Figure 7) requires 464 MB of memory, which is approx-

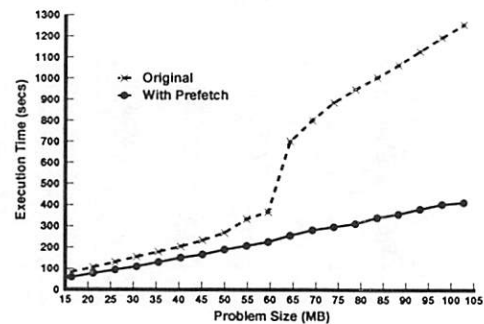


Figure 8: Performance of BUK (cold-started) across a range of problem sizes.

imately 10 times more than what is available. In all three cases, the performance improvements remain large. In fact, prefetching offers slightly larger speedup in all three cases since there is more I/O latency to hide.

4.3.3 Case Study: BUK

To illustrate the impact of I/O prefetching on performance across a wide range of problem sizes, we look at the BUK application as a case study. We chose BUK because we can easily set the problem size to any value for this application. Ignoring page faults, we would normally expect the execution time of BUK to increase linearly with the problem size. As we see in Figure 8, the original version of BUK (without prefetching) suffers a large discontinuity in execution time once the problem no longer fits in memory (recall that our prototype has 64 MB of physical memory, roughly 48 MB of which is available to the application). In contrast, the prefetching version of the code suffers

no such discontinuity—execution time continues to increase linearly. For this particular application, the prefetching version of the code outperforms the original code on all problem sizes, since even small problem sizes benefit from prefetching cold misses.⁶ Hence this application exemplifies what we are attempting to accomplish with automatic I/O prefetching: programmers can write their code in a natural manner and still achieve good performance even on out-of-core problem sizes.

5 Related Work

Much related work has depended on the use of an explicit I/O interface by the programmer. On the OS side, this work includes the automatic detection of file access patterns in the file system [1, 10, 11, 12, 14, 15, 17], as well as the use of access patterns supplied directly by the application using an I/O type of interface [22, 26, 30, 4]. For compilers it involves analysis to move explicit I/O calls back and change them to asynchronous I/O calls instead [25]. While some of the OS policies developed may be useful in our environment, our goal is to avoid the use of explicit I/O entirely.

Of the file system prefetching techniques mentioned above, the work on Transparent Informed Prefetching (TIP) by Patterson et. al [22] is most relevant to our work in that hints provided by the application level are used by the operating system to optimize file prefetching and replacement. In fact, the cost model employed by TIP might be very useful for our memory manager. However, TIP targets applications which are written to use explicit I/O, and they depend on the programmer (rather than the compiler) instrumenting the code with hints. Also, their hints follow a very different model, where no concept of time is embedded in the hints, and hence their operating system support must be more complex than ours.

Compiling for out-of-core array codes tends to focus on two areas. The first is reordering computation to improve data reuse and reduce the total I/O required [3]. The second area is inserting explicit I/O calls into array codes [6, 13, 21, 29]. In general, the compilers are aided by extensions to the source code that indicate particular structures are out-of-core. In addition, some of the work specifically targets I/O performance for parallel applications [3], while we have achieved impressive speedups for even single-threaded applications.

⁶For BUK, it is more realistic to cold-start the application, since it must always read its input data set from disk.

We feel that compiler analysis that targets an I/O interface is limited by the alias analysis problem described earlier, and in general cannot be as aggressive as an algorithm that supports non-binding prefetching.

Other work has also been done in the area of prefetching for paged virtual memory systems. As for file systems, some of the work depends on the OS detecting patterns to initiate prefetching [8, 27]. These techniques suffer from the fact that some number of faults are required to establish patterns before prefetching can begin, and when the patterns change unnecessary prefetches will occur. Using application-specific knowledge to assist memory management policies was studied by Malkawi and Patel [18], however they only considered retaining needed pages in memory and did not consider prefetching.

The most relevant work to our study was conducted nearly twenty years ago by Trivedi [32], who looked at the use application access patterns extracted by a compiler to implement “prepaging”. Although the interface to the OS is nearly identical, there are some significant differences. First, Trivedi’s compiler analysis was restricted to programs in which blocking could be performed whereas previous studies on prefetching for caches have shown that many programs which can be prefetched cannot be blocked [20]. Thus, our approach is much more widely applicable. Second, improvements in compiler analysis enable us to be much more aggressive, allowing the prefetching of indirect references and other interesting structures. Third, we have found that the use of the run-time layer is essential to achieving good performance when the compiler must deal with symbolic loop bounds, whereas this component was missing from earlier work.

6 Future Work

We view this work as an encouraging first step, and we are currently extending our research in the following directions. We are implementing our requisite support within commercial operating systems so that future results can be collected on larger, more modern systems where I/O latency is expected to be even more of a problem. To address the challenges of multiprogrammed workloads—where multiple applications compete for shared resources—we are exploring new ways that the compiler and OS can cooperate so that applications can adapt their behavior to dynamically fluctuating resource availability, and we will make more extensive

use of release operations to minimize memory consumption. Multiprocessors also provide interesting challenges, such as co-locating data (on disk) and computation within the same node to minimize network traffic. Page-based prefetching is applicable to domains other than disk I/O; for example, we are adapting our compiler technology to prefetch the page-sized chunks of data that are communicated between workstations in distributed shared memory (DSM) systems. Finally, we are investigating how to extend the scope of our work beyond array-based codes to also include pointer-based codes and other non-numeric applications.

7 Conclusions

This paper has demonstrated that with only minor modifications to current operating systems, we can enhance paged virtual memory to deliver high performance to out-of-core applications without placing any additional burden on the programmer. We have proposed and evaluated a fully-automatic scheme whereby the operating system and the compiler cooperate as follows: the compiler analyzes future access patterns to predict when page faults are likely to occur and when data is no longer needed, the operating system uses this information to manage I/O through non-binding *prefetch* and *release* hints, and a run-time layer interacts with the operating system to accelerate performance by adapting to dynamic behavior and minimizing prefetch overhead. We implemented our scheme in the context of a modern research compiler and operating system.

Our experimental results demonstrate that our scheme yields substantial performance improvements when we take unmodified, "in-core" versions of scientific applications and run them with out-of-core problem sizes. We successfully hid more than half of the I/O latency in all of the NAS Parallel benchmarks—in three cases, we eliminated over 98% of the latency. For five of the eight applications, this reduction in I/O stalls translates into speedups of roughly twofold, with two cases speeding up by threefold or more. Given these encouraging results, we advocate that commercial operating systems provide the modest support necessary for the *prefetch* and *release* operations.

8 Acknowledgments

We thank the entire Hurricane and Hector research teams for developing this experimental platform. In particular, we thank Ben Gamsa for answer-

ing countless questions about Hurricane's internal workings. We also thank Karen Reid, Eric Parsons, and Paul Lu for their help in collecting these results. This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

References

- [1] M. Arunachalam, A. Choudhary, and B. Rullman. A prefetching prototype for the parallel file system on the Paragon. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 321–323, May 1995. Extended Abstract.
- [2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.
- [3] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.
- [4] Pei Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. Technical report, November 1995.
- [5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [6] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.
- [7] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, December 1995.
- [8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.
- [9] J. M. del Rosario and A. Choudhary. High performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.
- [10] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.
- [11] A. S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Par-*

- allel and Distributed Information Systems*, page 177, December 1991.
- [12] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.
- [13] K. Kennedy, C. Koelbel, and M. Paleczny. Scalable I/O for out-of-core structures. Technical Report CRPC-TR93357-S, Center for Research on Parallel Computation, Rice University, November 1993. Updated August, 1994.
- [14] D. Kotz and C. Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.
- [15] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.
- [16] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, Philadelphia, May 1996.
- [17] T. M. Kroeger and D. D. E. Long. Predicting file system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 319–328, January 1996.
- [18] M. Malkawi and J. Patel. Compiler directed management policy for numerical programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 97–106, December 1985.
- [19] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.
- [20] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, pages 62–73, October 1992.
- [21] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, McLean, VA, February 1995.
- [22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.
- [23] J. T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.
- [24] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.
- [25] A. L. Narasimha Reddy, P. Banerjee, and D. K. Chen. Compiler support for parallel I/O operations. In *ICPP91*, pages II:290–II:291, 1991.
- [26] T.P. Singh and A. Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.
- [27] I. Song and Y. Cho. Page prefetching based on fault history. In *USENIX Mach III symposium proceedings*, pages 203–213, April 1993.
- [28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, , and G. Peck. Scalability in the XFS file system. In *USENIX Technical Conference*, pages 1–14. Usenix, January 1996.
- [29] R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *IPPS '94 Workshop on Input/Output in Parallel Computer Systems*, pages 54–72. Syracuse University, April 1994.
- [30] R. Thakur, R. Bordawekar, A. Choudhary, R. Pon-nusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.
- [31] S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992.
- [32] K.S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.
- [33] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.
- [34] Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector Multiprocessor. *IEEE Computer*, 24(1), January 1991.
- [35] D. Womble, D. Greenberg, R. Riesen, and S. Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.

A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching

Tracy Kimbrel* Andrew Tomkins† R. Hugo Patterson‡ Brian Bershad* Pei Cao§
Edward W. Felten¶ Garth A. Gibson† Anna R. Karlin* Kai Li¶

Abstract

High-performance I/O systems depend on prefetching and caching in order to deliver good performance to applications. These two techniques have generally been considered in isolation, even though there are significant interactions between them; a block prefetched too early reduces the effectiveness of the cache, while a block cached too long reduces the effectiveness of prefetching. In this paper we study the effects of several combined prefetching and caching strategies for systems with multiple disks. Using disk-accurate trace-driven simulation, we explore the performance characteristics of each of the algorithms in cases in which applications provide full advance knowledge of accesses using hints. Some of the strategies have been published with theoretical performance bounds, and some are components of systems that have been built. One is a new algorithm that combines the desirable characteristics of the others. We find that when performance is limited by I/O stalls, aggressive prefetching helps to alleviate the problem; that more conservative prefetching is appropriate when significant I/O stalls are not present; and that a single, simple strategy is capable of doing both.

1 Introduction

Recently there has been a great deal of interest in prefetching from parallel disks as a technique for improving the I/O performance of sequential applications. In this paper, we study prefetching and caching strategies for multiple disks in the presence of application-provided knowledge of future accesses. We compare the performance of four algorithms:

1. *Fixed horizon* is simple to implement, and has near-optimal performance when sufficient I/O parallelism is available, but can be suboptimal in I/O-bound situations.

2. *Aggressive* is also simple to implement, is close to optimal for a single disk and for well-laid-out data on multiple disks, but can be suboptimal for multiple disks when the load on the disks is unbalanced.
3. *Reverse aggressive* is substantially more complex, but is provably close to optimal for all configurations in a uniform fetch-time model of disk accesses.
4. *Forestall* is a new algorithm, representing an attempt to combine the desirable characteristics of the other three algorithms.

Using trace-driven simulation on a collection of file access traces, we compare the performance of these algorithms assuming an environment in which a single process is running and full advance knowledge is available.

1.1 Motivation

Our work is motivated by recent advances in technology that have made magnetic disks both cheaper and smaller. As a result, parallel disk arrays have become an attractive means for achieving high performance from storage devices at low cost [15, 28, 24]. Independently accessible multiple disks offer the advantage of both increased bandwidth and reduced contention on individual disk arms. However, many applications do not benefit from this parallelism because their I/O accesses are serial. This problem is particularly severe for read-intensive applications. Write performance is less important as write behind strategies can mask update latency. Read-intensive applications that stall for I/O a significant fraction of their running time include text search, 3D scientific visualization, relational database queries, multimedia servers and object code linkers.

Many of these applications have *predictable access patterns* [25, 1, 18]. The ability to provide the file system with hints about future references has motivated research into the design of policies that use this information to reduce I/O overhead [25, 26, 7, 6]. The two key techniques that are enabled by detailed information about future accesses are deep prefetching and better-than-LRU cache replacement.

This paper explores the tradeoff between aggressive prefetching and optimal cache replacement. The decision to

*Dept. of Computer Science and Engr., University of Washington, Seattle WA ({tracyk,bershad,karlin}@cs.washington.edu)

†School of Computer Science, Carnegie Mellon University, Pittsburgh PA ({andrewt,garth}@cs.cmu.edu)

‡Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA (rhp@cs.cmu.edu)

§Computer Sciences Dept., University of Wisconsin - Madison, Madison WI (cao@cs.wisc.edu)

¶Dept. of Computer Science, Princeton University, Princeton NJ ({felten,li}@cs.princeton.edu)

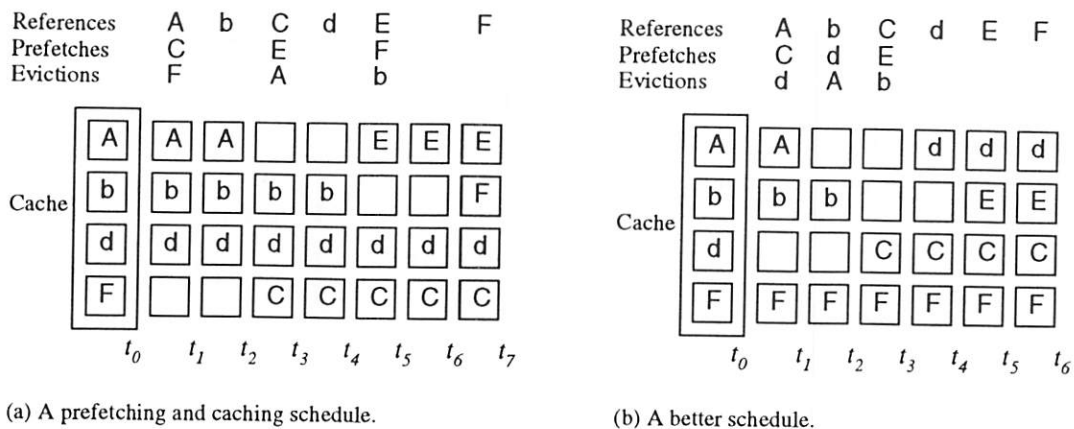


Figure 1: An example of prefetching and caching with advance knowledge of accesses for two disks. One disk holds blocks A, C, E, and F, and another disk holds blocks b and d. Cache size $k = 4$ disks and fetch time $F = 2$ computation steps.

prefetch requires that a buffer be reserved immediately, usually precipitating the replacement of cached data. This earlier replacement may result in an inferior replacement choice, which may actually increase the total number of fetches and degrade performance. Furthermore, in the multiple disk case, poor replacement choices can lead to load imbalance between the disks. The algorithms we study choose different points on the spectrum between aggressive prefetching (with possibly poorer cache replacements) and more conservative prefetching (with closer to optimal cache replacement).

The algorithms explored in this paper also differ in the degree to which they require advance knowledge. *Fixed horizon* exploits global knowledge the least while *reverse aggressive* exploits it the most. Using disk-accurate, trace-driven simulation, this paper's results provide a measure of the potential benefit of using global knowledge.

1.2 Parallel prefetching and caching

Prefetching and caching are even more complicated in a system with multiple disks, not only because it is possible to do multiple prefetches in parallel, but also because appropriate cache replacement strategies can alleviate load imbalance among the disks. Since a disk can serve only one prefetch at a time, a set of blocks can be prefetched in parallel only if they reside on different disks.

In order to develop intuition for why cache replacement strategies can affect parallel prefetching performance, consider the example shown in figure 1. In this example, the cache holds four blocks and the application references one block per time unit. If the application wants to reference a block that is not present in the cache, the application must wait or *stall* until the block is present. Suppose that it takes two time units to fetch a block from disk, and that the fetches on each disk are serialized. Every fetch evicts some block from the cache; the evicted block becomes unavailable at the moment the fetch starts. The goal is to minimize the total time spent by the application.

The application references blocks in the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d,

and F. Blocks A, C, E and F are on one disk, and blocks b and d are on the other disk. The straightforward approach is to prefetch aggressively: always fetch the missing block that will be referenced soonest; evict the block whose next reference is furthest in the future; but do not fetch if the evicted block will be referenced before the fetched block. For small caches such as in this figure, the fixed horizon and aggressive algorithms both behave in this way.

Figure 1(a) shows the cache block changes using this approach. The total elapsed time for the sequence is 7 time units. Figure 1(b) shows another prefetching schedule on the same reference pattern that is faster by one time unit. On the first fetch, d is evicted rather than F, even though d is referenced earlier. This has the advantage of offloading one fetch from the heavily loaded disk (the one holding A, C, E, and F) to the otherwise idle disk (the one holding b and d). This change allows two fetches to proceed in parallel later, thus saving one time unit.

The example shows that the achievable I/O parallelism of multiple disks can be affected by cache replacement and data placement policies. These are the factors that are addressed by the reverse aggressive algorithm.

1.3 Comparing approaches

The *fixed horizon* algorithm is based on the second Informed Prefetching (TIP2) system of Patterson, Gibson *et al.* [26], which manages allocation of cache space and I/O bandwidth between multiple processes, only some of which are disclosing some or all of their future accesses. TIP2 is designed for the case in which sufficient I/O bandwidth exists to service the request stream without stalling on I/O. The *fixed horizon* algorithm, a restriction of TIP2 to a single hinting process, initiates a fetch for a missing block H references ahead of its reference, where H is the ratio between the average time it takes to read a block from disk and the minimum time it takes to consume a block of data. Patterson *et al.* showed that under the assumption of sufficient bandwidth, this strategy eliminates stalls while placing little stress on system resources. However, *fixed horizon* will not look fur-

ther than H references into the future for fetches to perform. This can cause it to stall on I/O when there is insufficient disk bandwidth.

In contrast, the *aggressive* and *reverse aggressive* algorithms are designed to take maximum advantage of any amount of I/O parallelism. They use knowledge of future accesses to minimize application elapsed time for both small and large numbers of disks. *Aggressive* prefetches as early as possible, provided that the prefetched block is needed by the application sooner than the block that it will replace. When insufficient bandwidth is available, in particular, it becomes more important to schedule prefetch requests to ensure that no bandwidth is wasted.

Reverse aggressive goes beyond *aggressive*'s use of future knowledge by attempting to balance disk workload through carefully selected replacement decisions. Previously, it was shown theoretically that on any access pattern known in advance, *reverse aggressive*'s elapsed time is close to optimal [16]. It is the only one of these algorithms with this theoretical performance guarantee. However, it is not a practical algorithm. First, it is much more complex than the other algorithms, and second, its decisions depend on information farther in the future than the other algorithms. Nonetheless, its relative performance characteristics are of interest: we would like to understand whether or not the theoretical model we have defined actually gives insight into real system performance. If so, the theoretically near-optimal *reverse aggressive* can be used as a benchmark against which to compare other algorithms. The performance of *reverse aggressive* is our best *a priori* estimate of optimal performance.

The new algorithm *forestall* attempts to combine the best features of the other three algorithms: the good performance of *reverse aggressive* and the simplicity and implementability of *fixed horizon* and *aggressive*. *Forestall* avoids stalling while still making good (late) replacement decisions by estimating the point at which it needs to begin prefetching in order to prevent stalling.

1.4 Summary of results

In this paper we describe the results of a performance evaluation of the different policies for the d -disk integrated prefetching and caching problem. Our results from trace driven simulation demonstrate the practical performance characteristics of these algorithms. On our traces, we found that:

- All four algorithms significantly outperform demand fetching, even when advance knowledge of the access sequence is used to make optimal replacement decisions in conjunction with demand fetching.
- In compute-bound situations, *fixed horizon* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).
- In I/O-bound situations, *aggressive* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).

- In any given situation, one of *fixed horizon* or *aggressive* performs close to the theoretically near-optimal *reverse aggressive*.
- In all situations, *forestall* performs close to *reverse aggressive*.
- When data is well-laid out on the disks (e.g., striped), disk loads are balanced even without careful replacement choices. For this reason, *reverse aggressive* does not significantly outperform the other algorithms.
- *Fixed horizon* consistently places the least I/O load on the disks, due to its conservative fetching and near-optimal replacement choices. *Reverse aggressive* and *forestall* are intermediate between *aggressive* and *fixed horizon*.
- Batching of prefetch requests and disk head scheduling are crucial to the performance of prefetching and caching strategies.
- *Forestall* is a promising new approach that combines the best features of the other three algorithms: good performance regardless of I/O- or compute-boundedness, simplicity, and practicality.

We have focused on a rather narrow range of the input space: the single process, fully-hinted case. Clearly, prefetching and caching algorithms must deal effectively with missing or incorrect hints, as well as multiple simultaneously executing processes. *Fixed horizon*, *aggressive* and *forestall* can all be adapted to deal with these more general situations [5, 26].

1.5 Related work

Caching and prefetching have been known techniques to improve storage hierarchies for many years [2, 12]. In architectures, the work on caching and prefetching has focused on bridging the performance gap between CPU and main memory [29]. Research using caching and prefetching in database systems [9, 23, 10] showed that it is important to use applications' knowledge to perform caching and prefetching.

File caching and prefetching have become standard techniques for sequential file systems [12, 20, 14, 22, 30, 4, 13, 7, 26]. The most common prefetching approach is to perform sequential readahead [12, 20, 21]. The limitation of this approach is that it only benefits applications that make sequential references to large files. Another large body of work has been on predicting future access patterns [11, 30, 23, 10, 13]. Recently, caching and prefetching have also been studied for parallel file systems [11, 18, 25].

Although much work has been done in file caching and prefetching, most of it has considered one or the other in isolation. Recent studies for the single disk case showed [6, 5] that it is important to integrate prefetching, caching and disk scheduling together and that a properly integrated strategy can perform much better than a naive strategy. For the multi-disk case, a theoretical study [16] presented and analyzed *aggressive* and *reverse aggressive*. Other parallel prefetching strategies include one stripe lookahead prefetching on RAID

arrays, and Patterson *et al.* [26]’s TIP2 system. The one stripe lookahead benefits only applications that use large files, and would perform little prefetching for other applications. TIP2 uses the *fixed horizon* algorithm we have studied here. Patterson *et al.* [26] also present a cost-benefit technique for controlling buffer allocations for both hinting and non-hinting applications in a multi-process environment.

Previous studies of the algorithms considered here have been incomparable. Differences in hardware, both in the processor and the I/O system, as well as in the benchmarks used to evaluate the algorithms, have made it difficult to understand the differences between them. This paper represents the first direct comparison of these approaches. Using the knowledge learned from this comparison, we have designed a new algorithm that attempts to combine the best features of the previous efforts.

1.6 Organization of the paper

In the next section we describe the first three algorithms and their theoretical basis. In section 3, we describe our simulation framework. In section 4, we present the results of our simulations using the first three algorithms. In section 5 we describe the new algorithm *forestall* and present simulation results on its performance. We present our conclusions in section 6.

2 The algorithms

We begin by introducing the framework used to study this problem and the terminology used in the rest of the paper.

2.1 Theoretical model

Our theoretical model consists of two levels of memory hierarchy: a cache of K data blocks, and d (disk) storage devices. The execution of a program makes a known *request sequence* of references r_1, r_2, \dots, r_n to a set of m data blocks.

If a reference hits in the cache, it takes one time unit. Otherwise, the missed block must be fetched from a storage device. The system can either fetch a block on a miss (demand driven) or fetch the block before it is referenced in anticipation of a miss (prefetch). Either case takes F time units. If the cache is full, a cache block must be *evicted* before the fetch is issued to make room for the requested data block. While the fetch is in progress, neither the incoming block nor the discarded block is available for access.

We assume that each block resides on a single disk. Fetches to a single disk are serialized, but fetches on different disks can be executed concurrently.

When the program tries to access a block that is not in the cache, it stalls until the block arrives in the cache. The stall time is either F if the block is fetched on demand or $F - i$ if the block is prefetched i time units before the reference. The measure of performance is the elapsed time required to serve the entire request sequence; this is equal to the number of references plus the total stall time.

The goal is to minimize application elapsed time, by deciding when to fetch a block from a disk, which disk to fetch from, which block to fetch, and which cache block to evict (when the cache is full).

The time unit models the CPU time spent between two consecutive file references — the CPU time includes the time to copy the accessed file data from kernel address space to a user address space buffer, and the time for the application to consume the file data. The model simplifies the real situation by assuming that the CPU time between every two file references is the same, that all disk accesses take the same amount of time, and that there is no CPU overhead incurred by issuing an I/O request. These simplifications were made in order to make the problem theoretically tractable. Our simulations use actual CPU times collected in our traces and an accurate simulation model of modern disk drives, and charge a driver overhead for each request made to a disk.

2.2 Optimal prefetching rules

The following simple rules can be assumed of any optimal strategy in the single-disk case [6].¹

- *Optimal fetching*: when fetching, always fetch the missing block that will be referenced soonest;
- *Optimal replacement*: when fetching, always evict the block in the cache whose next reference is furthest in the future;
- *Do no harm*: never evict block A to fetch block B when A ’s next reference is before B ’s next reference;
- *First opportunity*: never evict A to fetch B when the same thing could have been done one time unit earlier.

Unfortunately, as exhibited in the example in section 1, some of these rules no longer hold in the multiple-disk case. It may be necessary to violate all of the rules except *first opportunity* to produce an optimal schedule.

2.3 The fixed horizon algorithm

As described earlier, the *fixed horizon* algorithm is based on the TIP2 system running a single hinting process [26].

Fixed horizon: Whenever there is a missing block at most H references in the future, issue a fetch for that block, replacing the cached block whose next reference is furthest in the future, provided that reference is further than H accesses in the future (which will certainly hold if $H < K$).

Fixed horizon is consistent with the first three rules of optimal prefetching for a single disk. An advantage of not following the fourth rule is that *fixed horizon* needs less information about references beyond the prefetch horizon than the other algorithms. A disadvantage is that when additional

¹These rules are optimal in the sense that any schedule that does not follow them can be transformed into one that does, with performance at least as good.

information is available, *fixed horizon* can have elapsed time nearly twice optimal.

The prefetch horizon H is computed as the ratio of the average time it takes to read a block from disk and the minimum time it takes to consume a single block of data. In the theoretical model, $H = F$.

Fixed horizon tries to fetch as late as possible without stalling in order to make the best possible replacement decision. Each fetch is issued so that it will complete just in time for the reference. If parallelism increases to the point that each request is made to an idle disk, this algorithm is optimal. However, in practice, a sufficient number of disks may not be available. In this case, *fixed horizon* may initiate fetches too late to avoid stalling. In fact, because it never initiates a fetch more than H references ahead of the missing block, *fixed horizon* may allow a disk to become idle even though the future requests beyond the prefetch horizon contain many missing blocks. On the other hand, if the missing blocks in the sequence tend to be separated by many intervening references to blocks that are present in the cache, we'd expect *fixed horizon* to have performance much closer to optimal than its worst case.

2.4 The aggressive algorithm

The (multi-disk) aggressive algorithm is based on the Cao *et al.* (single-disk) aggressive algorithm [6], which is provably near-optimal in the single-disk case.

(Multi-disk) aggressive: Whenever a disk is free, prefetch the first missing block on that disk, replacing the block whose next reference is furthest in the future, under the condition that the next access to the evicted block is after the next access to the block being fetched.

Aggressive is the most aggressive prefetching strategy that is consistent with the four optimal prefetching rules described in section 2.2. As mentioned, some of these rules are no longer valid in the multiple disk case. This provides some of the intuition for the following theorem.

Theorem 1 [16] *For any access pattern, and any layout of data on disks, the elapsed time of aggressive is at most $d(1+\epsilon_1)$ times that of the optimal elapsed time (the minimum possible), where d is the number of disks, and ϵ_1 is a small constant that depends on system parameters.²*

There are worst case access patterns/data layouts for which the elapsed time of aggressive is at least d times the minimum possible.

It is important to note that this worst case result depends on access patterns and data layouts in which the load is heavily unbalanced between the disks. If the request sequence is balanced, *aggressive* has near-optimal performance.

² ϵ_1 here is F/K where F is the fetch time/compute time ratio and K is the cache size measured in blocks. For typical system parameters ϵ_1 is less than 0.02.

2.5 The reverse aggressive algorithm

The *reverse aggressive* algorithm exploits global knowledge in order to produce a prefetching schedule that achieves near-optimal elapsed time. It does this by balancing disk workload through carefully selected replacement decisions.

Reverse aggressive: Construct a prefetching schedule for the *reversed* sequence that *replaces* at most one block on each disk in parallel as follows: Whenever a disk is free, determine the block B not needed for the longest time on that disk. If the next request to B is after the first missing block, issue a fetch for the missing block, *replacing* B . Transform this prefetching schedule back to a schedule for the original sequence by treating each fetch on the reverse sequence as an eviction on the forward sequence and vice versa.

For a proof of correctness, more details on how and why this algorithm works well, and a proof of the following theorem, see [16].

Theorem 2 [16] *For any request sequence, and for any layout of the data on the disks, the elapsed time of reverse aggressive is at most $1 + \epsilon_2$ times the optimal elapsed time.³*

There are two key properties of *reverse aggressive* that result in this theorem. First, whereas *aggressive* chooses evictions without considering the relative loads on the disks, *reverse aggressive* greedily evicts to as many disks as possible on the reverse sequence. In the forward direction, this translates to performing a maximal set of fetches in parallel. The fact that these are fetches in the forward direction means that at some point earlier in the sequence, corresponding blocks were evicted. Thus the eviction decisions of *reverse aggressive* on the forward sequence are based on the ability to prefetch the evicted blocks later on in parallel. Second, whereas *aggressive* can wastefully prefetch ahead on some of its disks, *reverse aggressive* is greedy in the reverse direction. Consequently, it is fetching blocks in the forward direction just in time (to the extent possible) for them to be used. This results in performing close to the best evictions possible for those fetches.

2.6 Practical considerations

Several important features of real systems are not captured by our theoretical model.

1. Disk response times and CPU times between I/O requests are not constant.

We use average values for each and expect that variation in event times does not substantially invalidate the algorithm's decisions. In our experimentation, this does not appear to be a major effect, with one exception (see section 4.3). (The systematic effects of disk scheduling on disk response time are considered separately).

³ ϵ_2 here is less than Fd/K , where F is the fetch time/compute time ratio, d is the number of disks, and K cache size in blocks. For typical system parameters, ϵ_2 is less than 0.1, and sometimes significantly less.

2. Access patterns exhibit locality of reference, and loads are balanced across the multiple disks when data is striped.

In practice, this allows both *fixed horizon* and *aggressive* to effectively utilize multiple disks, and achieve elapsed times comparable to the theoretically superior *reverse aggressive*.

3. Disk accesses require significant CPU overhead to form the request, communicate with the disk, and service the resulting interrupt(s). Thus, avoidable data fetches may add elapsed time even if they do not cause stalls.

Because the theory assumes that fetches entail no CPU overhead, this penalty punishes overly aggressive fetching. In practice, this effect favors the *fixed horizon* algorithm since its late replacement decisions tend to lead to the fewest fetches.

4. Disk response time is sensitive to the order in which requests are serviced.

In particular, disk scheduling reduces average disk response time as more accesses are presented and allowed to be reordered by the disk (driver). Although *fixed horizon* implicitly allows multiple outstanding requests at each disk, *aggressive* and *reverse aggressive* were defined to submit only one request at a time, since in the theoretical model there is no advantage to batching. Because of the significance of the disk scheduling effect, we modify the definitions of *aggressive* and *reverse aggressive* to submit disk requests in batches. We have found that the performance of all three algorithms benefits from the CSCAN disk scheduling algorithm.

Reverse aggressive also benefits from batching of requests during its construction of its prefetching schedule (the reverse pass over the request sequence). This is because typical request sequences exhibit spatial locality; by batching requests on the reverse pass, *reverse aggressive* generates missing blocks to be fetched on the forward sequence in groups that exhibit locality of reference.

The inter-request CPU time is actually composed of two components, a fixed amount of time to read a block out of the cache, and a variable amount of time to process the data. Our implementation of *fixed horizon* assumes the data processing time to be zero, and uses the ratio of the average disk response time to the time to read a block from the cache as the prefetch horizon H . This ensures that any prefetch to an idle disk will complete in time for the reference. Assuming an average disk response time of 15ms (which is usually an overestimate in our simulations) and 243μs to read a block from the cache (which was measured on the implemented TIP2 system) yields a value of $H = 62$; we used this value in all our simulations, except where noted otherwise.

2.7 Implementations of the algorithms

In the context of the considerations of the previous section, we summarize the implementations we compared.

Fixed horizon: Whenever there is a missing block at most H references away, issue a fetch for that block, replacing the block whose next reference is furthest in the future. Note that this algorithm may at any time have up to H outstanding references to a disk yielding opportunities for disk scheduling.

Aggressive: Whenever a disk D is free, construct a batch of at most *batch-size*⁴ fetches to initiate on D as follows: As long as the first missing block B on disk D precedes the block B' whose next request is furthest in the future, add the fetch/eviction pair B/B' to the batch. Issue the batch.

If two or more disks are free at the same time, we consider all their missing blocks together, in order of increasing request index. Each next missing block is issued to the appropriate disk (and the best possible choice of evictions is made), if the disk's batch is not full and the *do no harm* rule allows it. At some point, either the last free disk's batch becomes full or the *do no harm* rule disallows issuing further requests.

Reverse aggressive: Assuming a fixed ratio F between the time for a disk access and the inter-reference CPU time, consider the reversed sequence, and use it to derive a prefetching schedule as described in section 2.5, but construct the schedule in batches as done by *aggressive*.

This prefetching schedule is then transformed into a schedule of fetch/eviction pairs for the forward sequence. Associated with each eviction is a *release time*, the earliest index in the request sequence at which the block can be evicted (i.e. one greater than the index of the last request to the block until it is possibly fetched back into the cache at some later time.) The eviction choices are naturally ordered by increasing release point due to the method used by *reverse aggressive* to construct its schedule. Fetches may need to be re-ordered according to increasing request index; they are then matched to eviction choices according to these orderings.

This schedule is used to drive the disk model as follows. Whenever a disk D is free, add the first up to *batch-size* fetch/eviction pairs B_i/B'_i that have been released, and for which B_i resides on disk D , to the batch. Issue the batch.⁵

Notice that *aggressive* and *fixed horizon* use less lookahead information than *reverse aggressive*, in that for both of them, the "only" future information needed are the identities of the next missing blocks (up to H missing blocks for *fixed horizon*, and up to d times *batch-size* for *aggressive*), and their positions in the sequence relative to the next references to blocks currently in the cache.

3 Simulation framework

We used trace-driven simulation to evaluate the performance of the algorithms. We believe our simulation model to be an accurate reflection of the practical performance characteristics of the algorithms. The reference streams are taken from traces of real applications' behavior; the trace information

⁴The batch sizes used are listed in table 6.

⁵The batch sizes and estimate F used by *reverse aggressive* are discussed in section 4.4.

Sector size	sectors per track	tracks per cylinder
512 bytes	72	19
cylinders	rotational speed	disk cache size
1962	4002 rpm	128 Kbytes
ave. access time (8Kbyte)	controller interface	transfer rate
22.8ms	SCSI-II	10 MB/sec

Table 1: HP 97560 characteristics.

xds elapsed times (secs)				
	CMU simulator		UW simulator	
disks	F.H.	Agg.	F.H.	Agg.
1	63.3	61.6	65.6	63.7
2	36.9	34.1	38.0	34.3
3	33.6	33.9	36.2	33.7
4	33.8	35.1	34.2	35.1
5	33.0	34.2	33.5	34.4
synth elapsed times (secs)				
	CMU simulator		UW simulator	
disks	F.H.	Agg.	F.H.	Agg.
1	213.0	168.5	201.4	155.8
2	136.3	126.9	130.9	121.7
3	118.9	149.5	118.9	150.4
4	118.9	150.4	118.9	150.1

Table 2: Comparison of the simulators on the xds and synth traces.

we use is unaffected by prefetching and caching activity. The accurate modelling of disk fetch times, I/O driver overhead costs, and application process compute times in the simulations is a key difference relative to the theoretical framework. However, our simulators do not model serialization of DMA transactions.⁶

Two separate simulators were developed, one at Washington (UW) and one at Carnegie Mellon (CMU). The UW simulator uses the disk drive simulation of Kotz *et al.* [19] (which is based on that of Ruemmler and Wilkes [27]) to accurately model I/O costs. This simulation models fine architectural details to provide a very accurate simulation of the HP 97560 disk drive. Table 1 lists several characteristics of the HP 97560 (taken from [27]). The CMU simulator uses the Berkeley RaidSim [8] simulator, as modified at CMU, to simulate 0661 IBM Lightning disk drives.

The simulators were cross-validated on a common set of traces. The CMU simulator does not implement *reverse aggressive*. We obtained good agreement between the simulators on the results for *aggressive* and *fixed horizon* for several traces. Table 2 shows the elapsed times measured by the simulators for the xds and synth traces described below. Remaining differences between the simulators are consistent with the differences in the disk models. We report here results for all algorithms obtained using the UW simulator.

In our simulations, we ignore write operations. Write performance is less critical to I/O performance since the application generally does not have to wait for the disk to be

⁶We do not expect this to have a significant effect on the results since the DMA time is much less than the disk access time.

trace	reads	distinct blocks	compute time (sec)
dinero	8867	986	103.5
cscope1	8673	1073	24.9
cscope2	20206	2462	37.1
cscope3	30200	3910	74.1
glimpse	27981	5247	38.7
ld	5881	2882	8.2
postgres-join	8896	3793	11.5
postgres-select	5044	3085	79.2
xds	10435	5392	30.8
synth	100000	2000	99.9

Table 3: Trace summary data.

written. Moreover, the impact this has on the results is small since most of the references in our traces are reads.

We simulated disk arrays of sizes 1-8, 10, 12, and 16. Most of our figures show a smaller range of sizes, however. In each case, the performance with a larger number of disks is the same as that with the largest number of disks shown.

3.1 File access traces

We used a set of traces collected on a DECstation 5000/200. The running time of all the applications is dominated by disk read accesses. Each trace consists of a sequence of file block read requests in the order they were issued, and the sequence of measured process compute times between read requests, of a single execution thread. We used an I/O driver overhead of .5ms per I/O operation, which is typical of the 5000/200.

The applications are:

cscope[1-3]: an interactive C-source examination tool written by Joe Steffen, searching for eight symbols (cscope1) in a 18MB software package, searching for four text strings (cscope2) in the same 18MB software package, and searching for four text strings (cscope3) on a 10MB software package. With multiple queries, cscope will read multiple files sequentially multiple times.

dinero: a cache simulator written by Mark Hill. This application reads one file sequentially multiple times.

glimpse: a text information retrieval system from the University of Arizona, searching for four keywords in a 40MB snapshot of news articles. It builds approximate indexes for words to allow both relatively fast search and small index files. The result is that the index files are accessed repeatedly, whereas the data files are accessed infrequently.

postgres-join: the Postgres relational database system developed at the University of California at Berkeley, performing a join between an indexed 32MB relation and a non-indexed 3.2MB relation. The relations are those used in the Wisconsin Benchmark [3]. Since the result relation is small, most of the file accesses are reads. Here, the index blocks are accessed much more frequently than the data blocks.

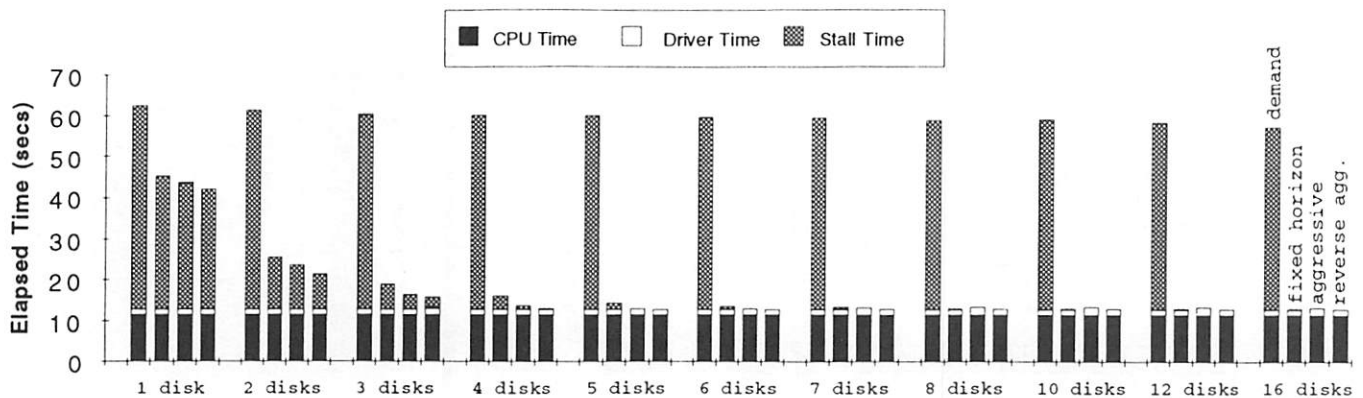


Figure 2: Performance on the postgres-select trace. Each group of bars represents the performance of the four algorithms *optimal demand fetching*, *fixed horizon*, *aggressive*, and *reverse aggressive*, in left-to-right order.

postgres-select: the Postgres relational database system executing a selection query of choosing 2% of the tuples from an indexed 32MB relation. The selection query is part of the Wisconsin Benchmark suite [3] and uses indexed search.

ld: the Ultrix link-editor, building the Ultrix 4.3 kernel from about 25MB of object files.

xds: a 3-D data visualization program, XDataSlice, generating 25 planar slice images at random orientations from a 64MB data file.

Finally, we used a synthetic trace **synth** containing 50 passes through a loop of 2000 sequential blocks. Compute times between read requests were generated according to a Poisson distribution with a 1 ms mean.

Table 3 shows the length (number of read requests), number of distinct blocks requested, and total application compute times for each of the traces.

The cache size was set to be 10MB (or $K = 1280$ blocks of 8 kbytes each) for all traces except *dinero* and *cscope1*. These traces contain references to fewer than 1280 distinct blocks. For these traces, the cache size was reduced to 4MB (512 blocks). We assume the cache to be empty (or to contain some other application's data) when the traced application starts. The entire cache is available to the traced application.

3.2 Data placement and disk head scheduling

The data was striped across the array using a one-block stripe unit. Some of our traces represented block numbers by (file,offset) pairs; for these we chose a random starting point within a group of 8550 8kbyte blocks (which occupy 100 cylinders on the HP 97560) for each file, corresponding to typical file system clustering mechanisms. The maximum seek time within a group of 100 cylinders is 7.24ms. Thus, in our simulations the average response time is typically lower than the 22.8ms listed in table 1. Other traces referred to logical filesystem block numbers; for these traces we used the actual block number for each access. Except where noted, we use CSCAN disk head scheduling.

4 Results

In the following sections, we examine the behaviors of the algorithms in detail. We begin by comparing the performance of the algorithms with that of demand fetching. We then examine the algorithms' performance on the synthetic trace, an easily understood access pattern that illustrates the key differences in behavior between the algorithms. Next we examine performance on the application traces, and explore the effects on the results of changes in various simulation parameters.

4.1 Comparison with demand fetching

In order to make this comparison as favorable as possible to demand fetching, we use the optimal offline replacement policy: whenever a block is fetched, the block in the cache whose next reference is furthest in the future is replaced. Figure 2 shows the elapsed times of the three algorithms and of optimal demand fetching on the postgres-select trace for varying numbers of disks between one and sixteen. The elapsed times are divided into three components: process compute time, I/O driver overhead (processor) time, and the time the processor spends idle, stalling on I/O. From this figure we see that (1) all three prefetching algorithms significantly outperform optimal demand fetching, and (2) the three prefetching algorithms achieve near linear reduction in I/O overhead until the applications become compute-bound. These two behaviors are consistent across all the applications we have studied.

4.2 Fundamental differences

The synthetic trace is used to examine the algorithms' behavior on a simple, known sequence in order to gain insight into the algorithms' performance. This trace shows the relative behaviors typical of the three algorithms in exaggerated form. Figure 3 summarizes the results for one to four disks.

The sequential accesses allow excellent performance from the disks; average response times are between 3 and 4 ms. In each case, *fixed horizon* performs 38000 fetches, 720 more

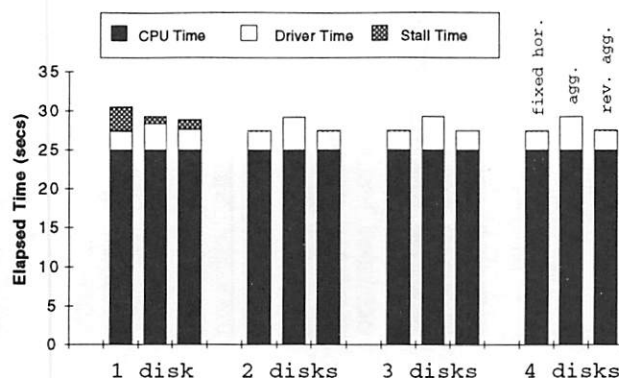
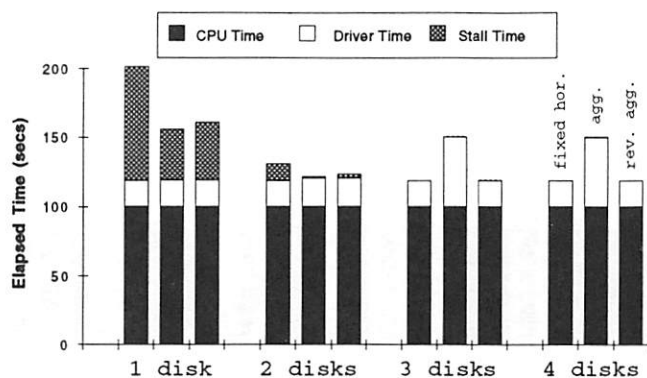


Figure 3: Performance on the synth (left) and cscope1 (right) traces. Each group of bars represents the performance of the three algorithms *fixed horizon*, *aggressive*, and *reverse aggressive*, in left-to-right order.

than the minimum possible 37280 performed by optimal demand fetching. (The total sequence length is 100,000).

With a single disk, the synthetic application is I/O bound. *Fixed horizon*'s conservative prefetching strategy reduces I/O stalling relative to demand fetching, but not as much as *aggressive*'s and *reverse aggressive*'s more aggressive strategies. After each pass through the loop under *fixed horizon*, the cache contains 1280 sequential blocks and the other 720 blocks in the sequence are not cached. The clustering of the 720 missing blocks allows good disk performance; however, the clustering of the 1280 cached blocks causes *fixed horizon* to leave the disk idle until the last H cached blocks are being read. *Aggressive* and *reverse aggressive* perform 39240 and 39265 fetches, respectively, slightly more fetches than *fixed horizon*'s 38000, resulting in a small difference in driver overhead. However, they are able to eliminate much of the I/O stall time by prefetching distant blocks and thus not idling the disk appreciably.

With two disks, *fixed horizon* is able to eliminate most of the stall time, without increasing the total number of fetches. *Aggressive* has nearly eliminated stall time completely, but at a higher driver cost due to its increased number (41902) of fetches. *Reverse aggressive* is between *fixed horizon* and *aggressive* in stall time; it performs 42000 fetches. Elapsed times are similar under all three algorithms. This case marks the transition from I/O-boundedness to compute-boundedness.

With three disks, stall time has been eliminated completely by all three algorithms. *Aggressive* uses the excess I/O bandwidth to prefetch and subsequently evict every block for every reference. In fact, because *aggressive* is willing to prefetch significantly ahead on one disk relative to others, it wastes 994 fetches, replacing a prefetched block from the cache before it is used in order to fetch a block on a different disk that will be needed sooner. Fortunately, this effect does not increase as the number of disks increases since with increasing I/O bandwidth, *aggressive*'s prefetching becomes so successful that every fetch is to the first missing block in the future. Such a block can never be replaced before it is used, since that would violate the *do-no-harm* rule.

Nonetheless, the elimination of stall time by *aggressive* comes at a high cost: the driver overhead for the extra fetches

pushes *aggressive*'s elapsed time higher than the two-disk case. In contrast, *fixed horizon* prefetches far enough ahead to serve all requests without stall, but no farther. Dedicating at most H buffers to prefetching, *fixed horizon* is able to eliminate stalling altogether without any additional fetches. *Reverse aggressive* performs 37907 fetches, fewer than *fixed horizon*, also eliminating stall time.

4.3 Application traces

The application traces show differences among the three algorithms similar to those shown by the synthetic trace, but less pronounced.

The right portion of figure 3 shows the performance of the three algorithms on the CPU-bound cscope1 trace. The behavior here is similar to that for the synthetic trace: *aggressive* eliminates stalling but issues too many fetches resulting in a greater driver overhead.

At the I/O-bound end of the spectrum, figure 4 shows a detailed breakdown of the performance of the three algorithms on the ld trace, from one to sixteen disks. With one disk, all three algorithms are I/O bound and have comparable performance. From two to eight disks, the more aggressive prefetching of *aggressive* and *reverse aggressive* results in somewhat less stalling than *fixed horizon*. At ten disks, *fixed horizon*'s performance matches *aggressive*'s. Beyond this point, the tradeoff between excessive stalling caused by leaving disks idle, and excessive driver overhead caused by prefetching aggressively, favors *fixed horizon* over *aggressive*. The other traces reflect similar trends, with different points of crossover: above five disks for postgres-select, glimpse, and cscope2, and below five disks for postgres-join, dinero, cscope1, and xds.

An exception to the generally best performance of *reverse aggressive* is the cscope3 trace, shown in figure 5. Note that *reverse aggressive*'s performance is much worse than *aggressive*'s with one disk. This is a case in which the differences between the theoretical model and the simulation model affect the performance of *reverse aggressive*. Recall that since *reverse aggressive* is offline, it generates a complete schedule based on its estimate of F . When it uses a smaller estimate of F , each fetch is assumed to complete earlier (relative to

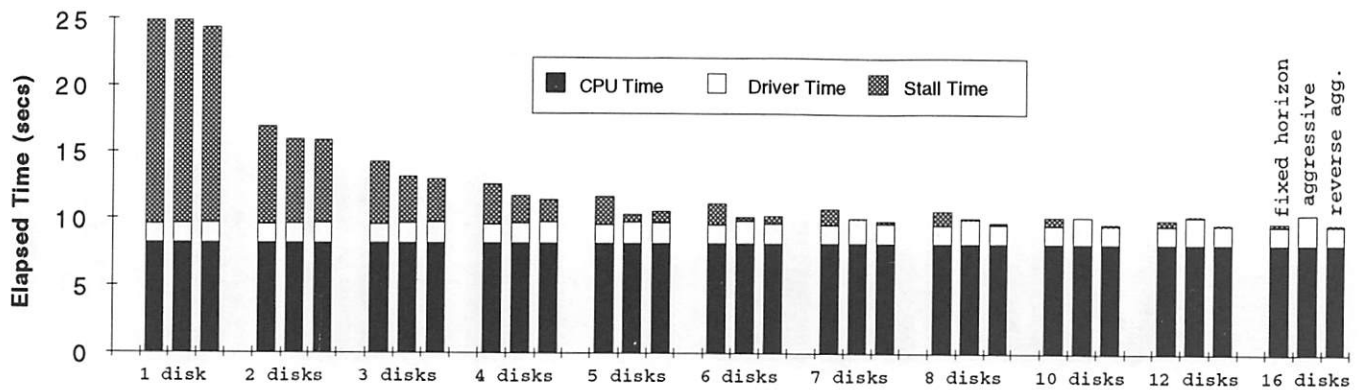


Figure 4: Performance on the ld trace.

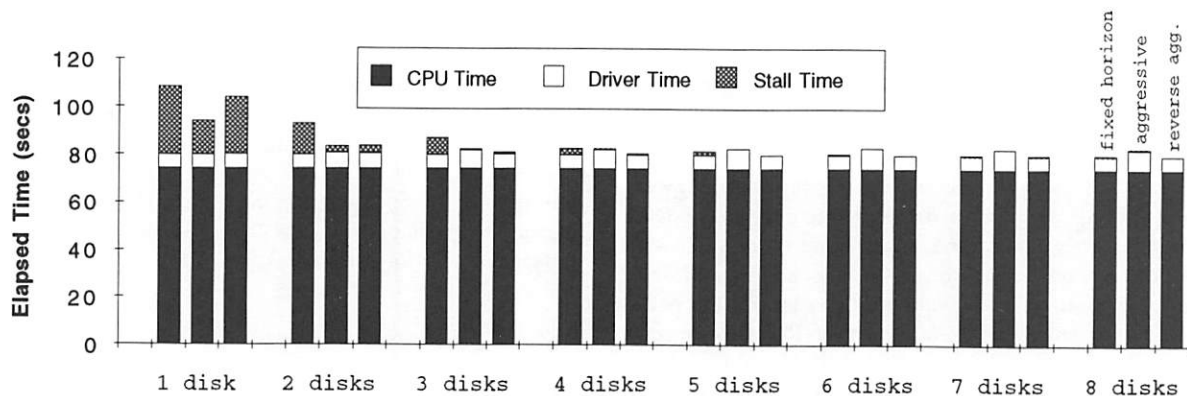


Figure 5: Performance on the cscope3 trace.

the inter-reference compute time) and therefore *reverse aggressive* generates a more aggressive prefetching schedule that keeps the disk(s) busier. When it uses a larger estimate of F , each fetch is assumed to take longer, and therefore *reverse aggressive* must delay the scheduling of subsequent fetches in the sequence, thus generating a more conservative prefetching schedule. In our implementation of *reverse aggressive*, the single best estimate of F is used for each trace. On traces with large variation in inter-reference compute times, any single estimate of F will be either too small or too large for some parts of the trace. This is the case for *cscope3* – examination of the trace reveals that the inter-reference compute times are bursty. Runs of compute times near 1ms are interspersed with runs of times around 7ms. Since the average fetch time on this trace with one disk is about 8ms, the ratio of fetch time to compute time (the “true” value of F) varies from about 1 to about 8.

In fact, with a single disk, *aggressive* has the same theoretical performance bounds as *reverse aggressive*. It is not surprising that *aggressive*’s inherent adaptivity to varying fetch times and compute times should give it an advantage over *reverse aggressive* in this case. This effect is noticeable, but less pronounced, on the *synth* trace as well.

On the remaining traces, *reverse aggressive*’s elapsed time varies from 3.6% worse to 10.7% better than the superior of *fixed horizon* and *aggressive* in any given configuration. For the full data, see [17].

disks	demand fetching	<i>fixed horizon</i>	<i>aggressive</i>	<i>reverse aggressive</i>
1	.82	.98	.99	.98
2	.41	.90	.92	.92
3	.27	.82	.87	.85
4	.20	.72	.81	.80
5	.16	.66	.70	.69
6	.13	.58	.63	.60
7	.12	.50	.62	.50
8	.10	.45	.56	.42
10	.08	.36	.43	.35
12	.07	.30	.35	.28
16	.05	.22	.26	.21

Table 4: Disk utilization on the postgres-select trace.

Table 4 shows the utilization of the disks (averaged over the disks when there are more than one) for demand fetching and the three prefetching algorithms on the postgres-select trace. For moderate numbers of disks, *aggressive* places the greatest load on the disks, followed by *reverse aggressive* and then *fixed horizon*; demand fetching places the least load on the disks. With a very high degree of disk parallelism, *reverse aggressive*’s offline schedule places even less load on the disks than *fixed horizon*’s conservative strategy.

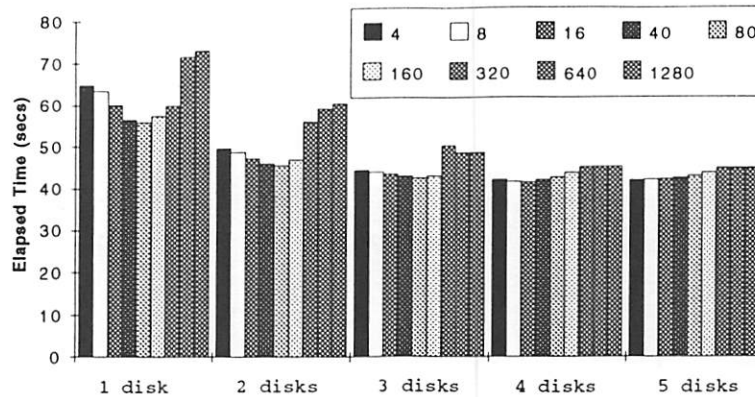


Figure 6: Performance of *aggressive* on the *cscope2* trace, as a function of the batch size.

4.4 Varying parameters

The performance of the algorithms depends on a set of parameters which interact in complicated ways with the applications' access patterns and inter-reference compute times, the layout of data on disks, the disk-scheduling discipline, and the characteristics of the disks. In this section, we explore the behavior of the algorithms when some of these parameters are varied. For lack of space, we present general observations and only a small portion of the data. For the full data, see [17].

We have already described most of the primary effects that explain what we see. These are:

- *scheduling*: an increase in the number of outstanding fetches issued by a prefetching algorithm results in increased latitude to reorder fetches and thus reduced disk response times. This effect is strongest in I/O-bound situations.
- *out-of-order fetching*: reordering of fetches can increase stall penalties when early missing blocks are fetched after later missing blocks. This effect is strongest in CPU-bound situations where any stall penalty is costly. When there is significant stalling, this effect is masked by other stalls and compensated for by the reduced average response time.
- *early replacement*: as prefetching becomes more aggressive, inferior replacement choices are made, leading to more fetches and in many cases, an increase in elapsed time.
- *limited aggressiveness*: the extent to which an algorithm can prefetch is limited by the *do no harm* rule.

Disk-head scheduling

The results shown in the previous section were obtained using CSCAN disk-head scheduling. CSCAN was used rather than SCAN since the HP 97560 contains a read-ahead buffer; CSCAN always scans in the same direction that the disk reads, improving the hit rate in the read-ahead buffer. We compared the performance impact of CSCAN

disks	<i>fixed horizon</i>	<i>aggressive</i>	<i>reverse aggressive</i>
1	14.9	19.2	24.0
2	4.85	11.3	22.1
3	2.59	8.36	19.9
4	0.53	3.59	6.71
5	-0.62	-0.77	0.0
6	-0.68	-0.31	0.0
7	-2.15	-0.45	0.0
8	-0.42	-0.17	0.0
10	-0.05	0.09	0.0
12	0.0	0.11	0.0
16	0.0	0.0	0.0

Table 5: Percentage improvement of CSCAN over FCFS on the *postgres-select* trace.

disk-head scheduling versus FCFS scheduling. Relative to FCFS, CSCAN improves the performance of *reverse aggressive* the most, up to 24%, and that of *fixed horizon* the least, up to 15%. For *aggressive*, the greatest benefit was 19%. Because of out-of-order fetching, CSCAN sometimes degrades performance slightly relative to FCFS in compute-bound situations. This effect is strongest for *fixed horizon* since it issues fetches later than they are issued by the other algorithms. The maximum degradation we observed is 3.6% (for *fixed horizon* with six disks on the *glimpse* trace).

Table 5 shows the performance benefit of CSCAN scheduling relative to FCFS on the *postgres-select* trace for all three algorithms with 1-16 disks.

The batch size used by *aggressive*

Figure 6 shows the effect of varying *aggressive*'s batch size on the *cscope2* trace. For each number of disks, performance initially improves with increasing batch size due to improved scheduling. For example, for one disk, the average fetch time drops from 10.4ms to 8.4ms as the batch size increases from 4 to 160. Eventually, out-of-order fetching and early replacement become more important and performance drops off again. For example, for one disk the number of fetches increases from 6771 to 9806 as the batch size increases from 160 to 1280.

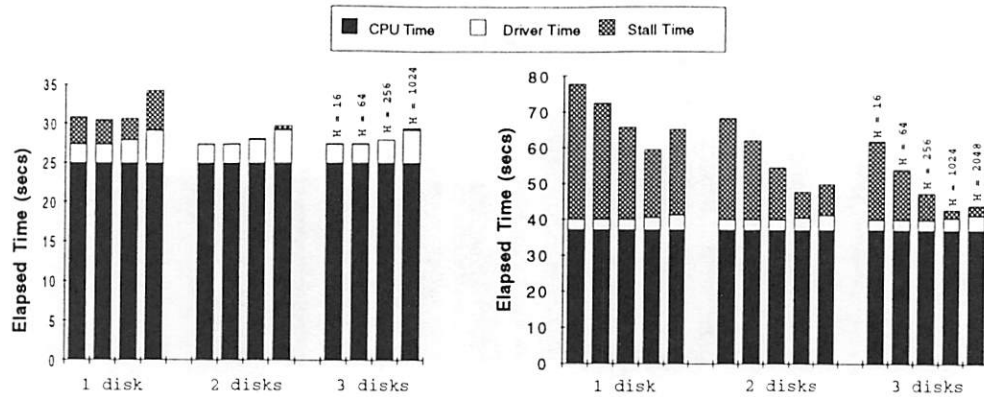


Figure 7: Performance of *fixed horizon* as a function of the prefetch horizon H on the *cscope1* (left) and *cscope2* (right) traces.

1 disk	2 disks	3 disks	4 disks
80	40	40	16
5 disks	6 disks	7 disks	> 7 disks
16	8	8	4

Table 6: Batch sizes used for *aggressive*.

As the number of disks increases, the variation in performance with batch size diminishes, and the best batch size shifts to a smaller value. This is because in more compute-bound situations, out-of-order fetching and limited aggressiveness are the dominant factors. Because of limited aggressiveness, the number of fetches increases only from 11325 to 11399 as batch size increases from 160 to 1280 with 5 disks.

Although the optimal batch size decreases with the number of disks for all the traces, it varies significantly from trace to trace. For example, for the *xds* trace, the optimal batch size for one to three disks was 16, and for four or more was 4. All the results for *aggressive* presented in section 4.3 were obtained using the batch sizes given in table 6. The performance of *aggressive* with these fixed batch sizes is on average 0.7 % worse (and at most 11% worse) than its performance with the best batch size for the configuration.

Prefetch horizon

The left side of figure 7 shows the effect of varying *fixed horizon*'s prefetch horizon H on the *cscope1* trace. We see that for each number of disks, performance deteriorates with increasing H (except on one disk, where it improves slightly until $H = 64$ is reached). This is due to out-of-order fetching and early replacement. For example, with 1 disk, earlier replacements cause the number of fetches to increase from 4959 with $H = 64$ to 8535 with $H = 2048$. Out-of-order fetching accounts for all the stall time with 2 and 3 disks when $H \geq 512$; using FCFS scheduling this stall time is eliminated.

On the more I/O bound traces such as *cscope2*, also shown in figure 7, we find a significant initial performance improvement with increasing H because the more aggressive prefetching eliminates stalling. Only at very large values of H does performance decline again.

The parameters used by *reverse aggressive*

We experimented with the batch size and fixed value of F used by *reverse aggressive* to construct its schedule on its reverse pass over the request sequence, as well as the batch size used on the forward pass. Since we use *reverse aggressive* only as a benchmark against which to compare the other algorithms, the main purpose of these experiments was to determine the optimal configuration (choice of F and batch sizes) for each trace, for each number of disks.

These experiments show that, as with *aggressive*, a smaller (resp. larger) batch size benefits a more compute-bound (resp. I/O-bound) application. Recalling that as *reverse aggressive*'s estimate of F decreases, it becomes increasingly aggressive, we similarly find that a smaller (resp. larger) value of F benefits a more I/O-bound (resp. compute-bound) application.

Processor speed and cache size

In order to assess the impact of improved CPU performance relative to disk performance, we ran our trace-driven simulations assuming a processor twice as fast. For these tests, *fixed horizon*'s prefetch horizon H was doubled to 124. The results are entirely unsurprising: faster processors are more dependent on I/O performance so that the payoff of using multiple disks and prefetching is increased. In addition, since a larger number of disks is needed to eliminate I/O overhead, the point at which the tradeoffs begin to favor *fixed horizon* over *aggressive* is shifted to a larger number of disks. This behavior was consistent across the applications.

In order to assess the impact of cache size on performance, we ran our trace-driven simulations with varying cache sizes: 640, 1280, and 1920 blocks. As cache size increases, the performance of all the algorithms improves. In I/O-bound cases, a larger cache improves *aggressive*'s and *reverse aggressive*'s performance more than *fixed horizon*'s since they prefetch more aggressively. In more compute-bound cases, *aggressive*'s excessive driver overhead penalizes it even more with a larger cache, so that *fixed horizon*'s performance relative to *aggressive* improves slightly as cache size increases. This is illustrated in table 7, which shows the performance of *fixed horizon* relative to *aggressive* as percentage difference, as a

cache size	1 disk	2 disks	4 disks	8 disks	16 disks
640	6.0	14.7	24.8	7.3	-2.6
1280	11.3	20.2	24.5	5.7	-3.8
1920	13.8	25.0	21.7	5.7	-3.8

Table 7: Elapsed time as a function of the cache size and number of disks of *fixed horizon* relative to *aggressive* (percentage difference) on the glimpse trace.

function of the cache size and the number of disks on the glimpse trace.

5 A new approach

We have designed a new algorithm, *forestall*, attempting to combine the best features of all three previously described algorithms: the good performance of *reverse aggressive* regardless of I/O-boundedness or compute-boundedness, and the simplicity and implementability of *fixed horizon* and *aggressive*. *Forestall* tries to avoid stalling while still making good (late) replacement decisions by estimating the point at which it needs to begin prefetching in order to prevent stalling. It makes this estimate based on its current cache state.

Returning to the theoretical model, suppose that there is a single disk, and that at some point during the servicing of the request sequence, the cache contains the next $2F - 1$ blocks requested. (Recall that in the theoretical model, the interference CPU time is taken to be 1 time unit, and the time to fetch a block from disk is F time units.) Further suppose that the subsequent two requests are missing from the cache. *Aggressive* immediately starts fetching and avoids stalling on the missing blocks, bringing the second missing block into the cache at time $2F$ — just in time to serve the request without stalling. *Fixed horizon* leaves its disk idle until the cursor is within F requests of the first missing block; it stalls $F - 1$ steps on the second missing block. In contrast, suppose there is only one missing block at a distance of $2F - 1$ from the cursor. In this case, *aggressive* will fetch immediately and make a possibly inferior replacement choice. *Fixed horizon* waits until its cursor is within F steps of the missing block, and prefetches just early enough to avoid stalling; in the intervening time, it may have finished using a block that is not needed until later in the sequence (if at all) than the one evicted from the cache by *aggressive*.

Forestall behaves as does *aggressive* in the first case, and as does *fixed horizon* in the second. For each i , $i \geq 1$, let d_i denote the distance from the cursor to the i^{th} missing block in the request sequence. For any $i \geq 1$, if $iF > d_i$, processing will surely stall on the i^{th} missing block or some earlier missing block. It will take iF time units to fetch the first i missing blocks, and at most the next d_i requests can be served concurrently. *Forestall* initiates a prefetch according to the *optimal fetching* and *optimal replacement* rules whenever $iF \geq d_i$ is true for some i and the *do no harm* rule allows it.

Practical considerations

As do the other algorithms, *forestall* requires modifications in order to account for differences between the theoretical model and real systems. Requests need to be issued in batches in order to reduce average disk access times. The ratio F of disk response time to interaccess time is not constant and must be estimated. In our implementation, we estimate F by tracking recent disk response times and compute times: F is dynamically computed on a per-disk basis as the ratio between the sum of the most recent 100 disk access times and the most recent 100 interreference CPU times.

Just as we needed the prefetch horizon H to be an overestimate of F for *fixed horizon* to have adequate performance, *forestall*'s performance depends on overestimating F in certain situations as well. We denote by F' the overestimate of F used by *forestall*. We evaluated *forestall*'s performance with different values of the parameter F' . We found that the best choice of F' depended on the per-trace average disk access times. For those traces for which the average disk access time was small, in the 3-4ms range, it was best to take $F' = F$. For those traces for which the average disk access time was larger, it was best to take $F' = 4F$. This is not hard to explain. Traces with disk access times in the 3-4ms range must contain a great deal of sequential access, so that most requests hit in the disk's readahead cache and are served by the CSCAN scheduler in the order in which they are received. When this happens, it is not necessary to prefetch aggressively. When the disk access times are large, the access pattern is more complicated, and disk access times more varied. *Forestall*'s mechanism for deciding when to prefetch benefits from overestimating the potential to stall. This smooths out the variations and avoids stalling due to the reordering of requests by CSCAN. Our implementation of *forestall* adapts to the observed disk access times, using the small value of F' for small disk access times (less than 5ms on average), and the larger value of F' for larger disk access times. Finally, because of the reordering of requests by CSCAN, we found it necessary to add *fixed horizon*'s rule to issue a fetch whenever the cursor is within H requests of a missing block. This avoids stalling on reordered requests in situations in which the $iF' \geq d_i$ rule delays fetching until the cursor is very near the first missing block.

Rather than using complete lookahead information in our implementation of *forestall*, we check the value of the expression $iF' - d_i$ only for those missing blocks within distance $2K$ of the cursor, where K is the cache size. We have not experimented with different values of this parameter, nor with variations of the history length 100 used to track fetch times and application process compute times.

Forestall's dependence on batch-size is similar to *aggressive*'s. We used for *forestall* the batch sizes given in table 6.

To compare static vs. dynamic estimation of *forestall*'s parameter F' , we compared the performance of *forestall* using fixed values of 1, 2, 4, 8, 15, 30, and 60 for F' to its performance using the dynamic estimation just described. Because actual average inter-reference compute times in our traces vary greatly (from 1.3ms for postgres-join to 15.7ms for the postgres-select), no single value can work well for all traces.

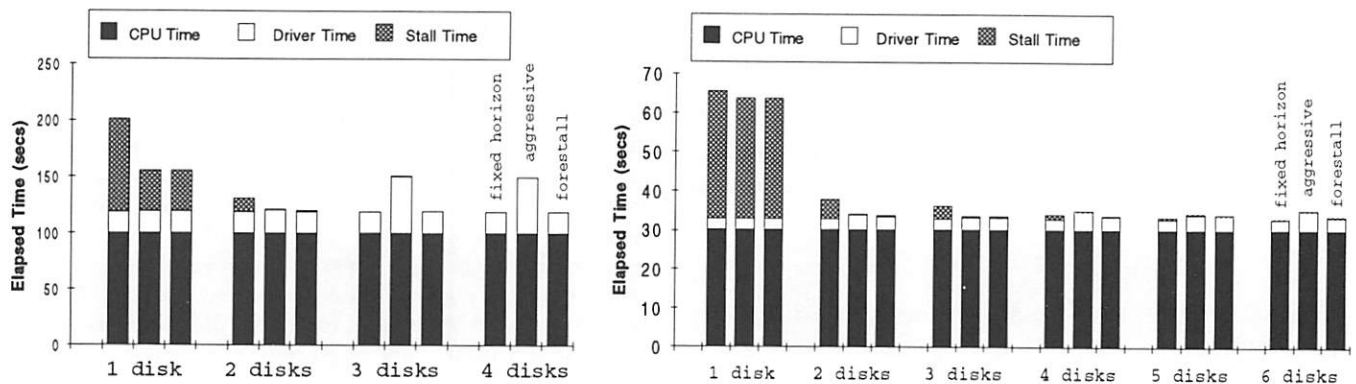


Figure 8: Performance on the synth (left) and xds (right) traces. Each group of bars represents the performance of the three algorithms *fixed horizon*, *aggressive*, and *forestall*, in left-to-right order.

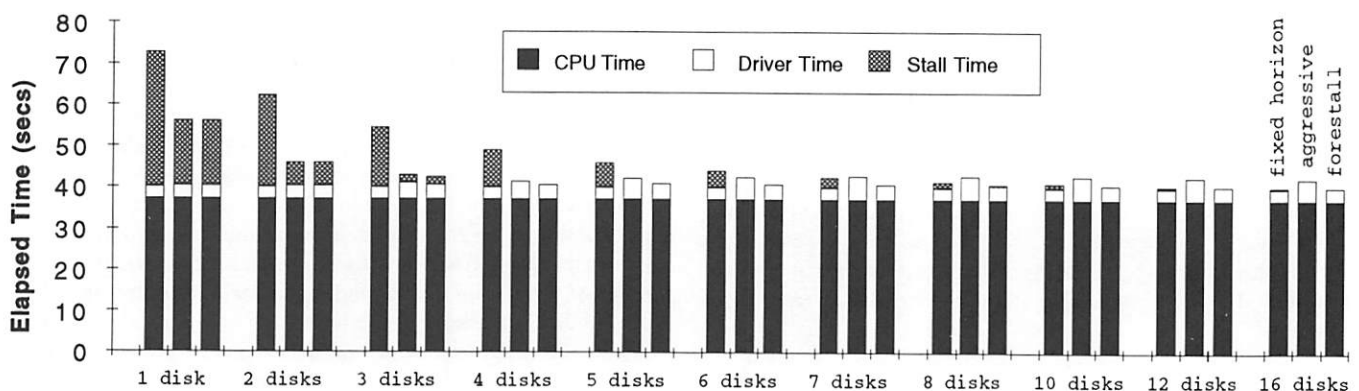


Figure 9: Performance on the cscope2 trace.

The values with the least maximum degradation relative to the dynamic algorithm, over all traces and disk array sizes, are 30 and 60; performance is at most 6.8% worse than the dynamic estimation (on the j2 trace with 2-6 disks). We exclude the synth trace from this calculation, since its artificially low disk response times demand a very low value of F' . For each trace, there is a fixed choice of F' that works well across all disk array sizes. This best choice varies from 1 for the dinero trace to 60 for the glimpse trace. The maximum degradation relative to the dynamic estimation allowing this much flexibility is 1.4% (for the ld trace with 7 disks and $F' = 30$). Finally, if we choose the best fixed value for each trace and each disk array size, the maximum degradation relative to the dynamic estimation is 1.2% for the cscope1 trace with one disk and $F' = 2$; for all other traces and array sizes, as well as all other array sizes for this trace, the degradation is less than .5%.

These results indicate that choosing the right parameters between workloads is more important than choosing the right parameter within a particular workload. Furthermore, *forestall*'s performance even with a single fixed parameter over all workloads and array sizes is always within 7% of optimal, and is almost always within 4% of optimal. This suggests that the advantages of *forestall* are due to its estimation of and adaptivity to upcoming disk load rather than the dynamic nature of its fetch-time and compute-time estimates.

5.1 Performance of *forestall*

Figure 8 shows the performance of the three practical algorithms, *fixed horizon*, *aggressive* and *forestall*, on the synthetic trace and xds. *Forestall* behaves exactly as expected. In the I/O bound situations, it prefetches aggressively enough to perform as well as or even better than *aggressive*. In the CPU-bound situations, it becomes more conservative in its prefetching, and has a lower driver overhead, matching the performance of *fixed horizon*.

Figures 9 and 10 show the performance of the three algorithms on the cscope2 and glimpse traces. Once again, *forestall* has the best performance of the three practical algorithms. On all remaining traces, over all configurations, *forestall*'s performance was between 2% worse and 5.8% better than the best of *aggressive* and *fixed horizon* in that configuration. For the full data, see [17].

Table 8 shows the utilization of the disks by *forestall* on the postgres-select trace. Its utilization falls between those of *aggressive* and *fixed horizon*, as expected. Moreover, in I/O-bound situations, it places a load on the disks similar to *aggressive*'s; in compute-bound situations, it places a lower load on the disks, similar to that of *fixed horizon*.

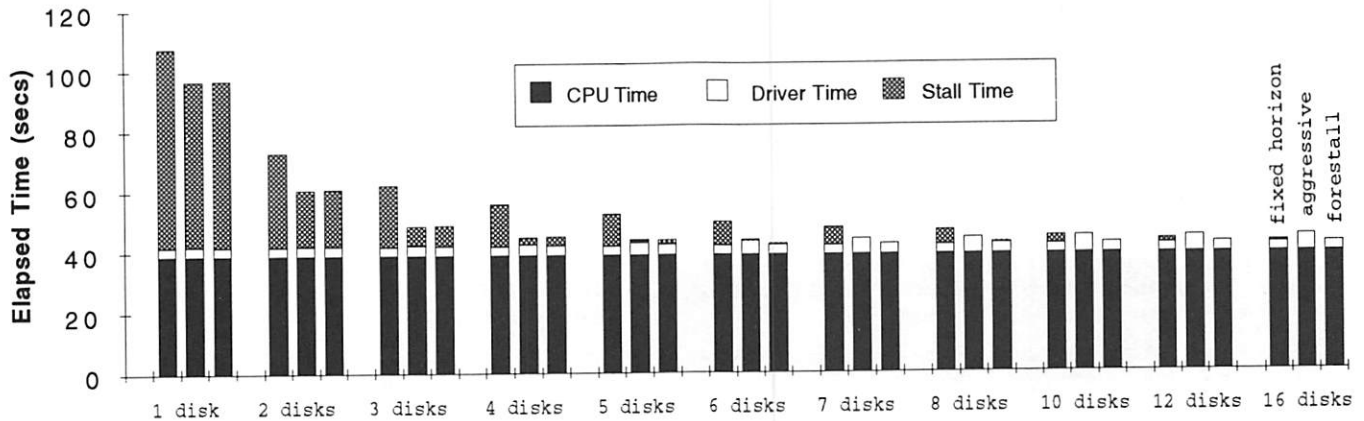


Figure 10: Performance on the glimpse trace.

disks	1	2	3	4	5	6
util.	.99	.92	.87	.81	.68	.63
disks	7	8	10	12	16	
util.	.62	.54	.39	.30	.32	

Table 8: Utilization of disks by *forestall* on the postgres-select trace.

6 Conclusions

This paper presents the results of a trace-driven simulation study of integrated prefetching and caching algorithms on a single read-only access sequence, assuming that all accesses are known in advance. We studied four algorithms: *aggressive*, *fixed horizon*, *reverse aggressive*, and *forestall*. We found that the theoretically near-optimal *reverse aggressive* usually has the best performance of the four algorithms, but that, perhaps surprisingly, it was never much better than the best of the other algorithms. This shows that carefully choosing replacements is not necessary to balance the load across the disks when the data is well laid out. We found that each of *aggressive* and *fixed horizon* performs well under the conditions for which it was designed, and in any given situation, one or the other performs similarly to *reverse aggressive*. Clearly, *aggressive* and *fixed horizon* are much more practical algorithms than *reverse aggressive*. These observations led us to the hybrid approach of *forestall*, which prefetches more aggressively in I/O-bound situations and more conservatively in compute-bound situations, resulting in nearly the best performance of the four in all configurations.

This study leaves several important issues unresolved. The performance of the algorithms depends on a set of parameters which interact in a complicated way with the applications' access patterns and inter-reference compute times, the layout of data on disks, the disk-scheduling discipline, and the characteristics of the disks. At this time, we have no analytical basis for dynamically determining *aggressive*'s batch size, *fixed horizon*'s prefetch horizon H , *reverse aggressive*'s batch sizes and estimate of F , or *forestall*'s batch size and estimate F' of F . It is a challenging open problem to fully understand the interaction between the algorithmic parameters and the specific application and system charac-

teristics.

Another direction for future research is the treatment of writes, both theoretically and experimentally.

We have not considered the effects of incomplete or inaccurate hints and we have not dealt with the question of how to allocate buffers among competing processes. While the three practical prefetching algorithms can easily be adapted to deal with these situations ([5, 26]), we expect differences in their performance. Aggressive prefetching increases both disk utilization and cache utilization. Therefore, disks are more likely to be busy when unhinted accesses occur. Moreover, an aggressively prefetching process might consume too large a fraction of the cache relative to a nonhinting process. Since *fixed horizon* places the least load on the disks and the cache, it is likely to be least affected by unhinted accesses and to have the smallest impact on other executing processes.

Lastly, this work reaffirms that the operating system can effectively take advantage of hints. An important research direction is to determine how applications can easily provide such hints.

Acknowledgements

Tracy Kimbrel and Anna Karlin wish to thank Martin Tompa for his continued encouragement and advice. We could not have managed the production of this document without the help of Dylan McNamee. Karin Petersen, our paper shepherd, was of great help in improving the quality of presentation.

This research is supported in part by NSF grant numbers ECD-8907068, CCR-9301186, GER-9450075, CCR-9632769, in part by DARPA Contract numbers DABT63-94-C-0049, DABT63-93-C-0054, in part by generous contributions from the member companies of the Parallel Data Consortium, and in part by Intel Corporation and Digital Equipment Corporation. Tracy Kimbrel is supported by an Intel Foundation Graduate Fellowship. Brian Bershad is supported by an NSF Presidential Faculty Fellowship. Ed Felten is supported by an NSF National Young Investigator Award. The views and conclusions contained in this document are those of the au-

thors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

References

- [1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz and A. Sussman. Tuning the Performance of I/O-Intensive Parallel Applications. *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, pages 15–27, May, 1996.
- [2] L.A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.
- [3] Jim Gray. *The Benchmark Handbook*. Morgan-Kaufman, San Mateo, CA. 1991.
- [4] Pei Cao, Edward Felten, and Kai Li. Application-Controlled File Caching Policies. In *USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.
- [5] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. Technical Report TR-CS95-493, Princeton University, 1995.
- [6] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of Integrated Prefetching and Caching Strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages , May 1995.
- [7] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, November 1994.
- [8] P.M. Chen and D.A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 322–331, May 1990.
- [9] H.T. Chou and D.J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 127–141, Dublin, Ireland, 1993.
- [10] Kenneth M. Curewitz, P. Krishnan, and Jeffrey S. Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, pages 257–266, Washington, DC, May 1993.
- [11] Carla Schlatter Ellis and David Kotz. Prefetching in File System for MIMD Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 306–314, August 1989.
- [12] R.J. Feiertag and E.I. Organisk. The Multics Input/Output System. In *Proceedings of the 3rd Symposium on Operating Systems Principles*, pages 35–41, 1971.
- [13] Jim Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.
- [14] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [15] M. Kim. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, 35(11):978–988, 1986.
- [16] Tracy Kimbrel and Anna R. Karlin. Near-optimal Parallel Prefetching and Caching. In *Proceedings of the 1996 IEEE Symposium on Foundations of Computer Science*, October 1996.
- [17] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. Technical Report UW-CSE-96-09-01, University of Washington, 1996.
- [18] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.
- [19] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report PCS-TR94-220, Department of Computer Science, Dartmouth College, July 1994.
- [20] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [21] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, 1991.
- [22] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [23] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September 1991.
- [24] D.A. Patterson, G. Gibson, and R.H. Katz. A Case for Redundant Arrays for Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD Conference*, pages 109–116, June 1988.
- [25] R. Hugo Patterson and Garth A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.
- [26] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95, December 1995.
- [27] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modelling. In *IEEE Computer*, 27(3):17–28, March 1994.
- [28] K. Salem and H. Garcia-Molina. Disk Striping. In *the 2nd IEEE Conference on Data Engineering*, pages 336–342, Feb. 1986.
- [29] Alan J. Smith. Second Bibliography on Cache Memories. *Computer Architecture News*, 19(4):154–182, June 1991.
- [30] C. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *Proceedings of Parallel and Distributed Information Systems*, pages 190–196. IEEE, 1991.

Efficient Cooperative Caching using Hints

Prasenjit Sarkar and John Hartman
{psarkar,jhh}@cs.arizona.edu
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

We present a very low-overhead decentralized algorithm for cooperative caching that provides performance comparable to that of existing centralized algorithms. Unlike existing algorithms that rely on centralized control of cache functions, our algorithm uses *hints* (i.e. inexact information) to allow clients to perform these functions in a decentralized fashion. This paper shows that a hint-based system performs as well as a more tightly coordinated system while requiring less overhead. Simulations show that the block access times of our system are as good as those of the existing tightly-coordinated algorithms, while reducing manager load by more than a factor of 15, block lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

1 Introduction

Caching is a common technique for improving the performance of distributed file systems [Howard88, Nelson93, Sandberg85]. Client caches filter application I/O requests to avoid network and server traffic, while server caches filter client cache misses to reduce disk accesses. A drawback of this organization is that the server cache must be large enough to filter most client cache misses, otherwise costly disk accesses will dominate system performance.

A solution is to add another level to the storage hierarchy, one that allows a client to access blocks cached by other clients. This technique is known as *cooperative caching* [Dahlin94] and it reduces the load on the server by allowing some local client cache misses to be handled by other clients.

The *cooperative cache* differs from the other levels of the storage hierarchy in that it is distributed across the clients and it therefore shares the same physical memory as the local caches of the clients. A local client cache is controlled by the client, and a server cache is controlled by the server, but it is not clear who should control the cooperative cache. For the cooperative cache to be effective,

the clients must somehow coordinate their actions.

The major contribution of this paper is to show that a cooperative caching system that relies on local hints to manage the cooperative cache performs as well as a more tightly-coordinated fact-based system. Our motivation is simple: hints are less expensive to maintain than facts and as long as hints are highly accurate, they will improve system performance. Hence, instead of maintaining and accessing global state, each client in a hint-based system gathers its own information about the cooperative cache. Thus the key challenge in designing a hint-based system is to ensure that the hints are highly accurate with minimal overhead.

In this paper, we describe a cooperative caching system that uses hints instead of facts whenever possible. We then use trace-driven simulation to show that the block access times of the proposed hint-based algorithm are as good as those of the existing tightly-coordinated algorithms, while reducing manager load by more than a factor of 15, lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

The remainder of the paper is organized as follows: Section 2 introduces the issues in cooperative caching. Section 3 briefly discusses previous cooperative caching algorithms. Section 4 describes the proposed hint-based algorithm. Section 5 describes our methodology for evaluating the algorithms, and Section 6 evaluates the performance of all the algorithms and analyzes the tradeoffs in design. Related work is discussed in Section 7 and we conclude in Section 8.

2 Cooperative Caching

A cooperative caching algorithm involves three logical entities: *clients*, *servers*, and *managers*. Clients access the blocks stored on the servers, and the managers control the cooperative cache. The control provided by the managers may include locating blocks in the cache and deciding which blocks to cache, but it varies among cooperative caching algorithms and serves as the major distinction

between them.

Although it is enticing to think of cooperative caching as simply another layer in the storage hierarchy, management of the cooperative cache can potentially involve every machine in the system since the cache is distributed across all the clients. The following sections look at the coordination aspects of cooperative caching and discuss design approaches.

2.1 Block Lookup

Handling a local cache miss by a client requires retrieving the missing block from the lower levels of the storage hierarchy. This procedure is called *block lookup* and in cooperative caching, it involves retrieving blocks from the cooperative cache. However, the cooperative cache is distributed over the caches of all the clients. Thus, retrieving a block from the cooperative cache would require information about the location of blocks in the client caches.

There are two possible strategies of maintaining block location information. One strategy is based on managers which keep track of which blocks are present in the client caches. A client does a block lookup by sending a request to the manager responsible for the block. To ensure that managers have up-to-date block location information, all block movement in and out of the client caches must be reported to the managers.

An alternative strategy is to let the clients maintain block location hints and do the block lookup themselves, avoiding the cost of contacting managers.

2.2 Replacement Policy

The role of the cooperative cache replacement policy is to determine the order in which blocks are replaced. The cooperative cache replacement policy is activated when a client decides to replace a block from its local cache. This block can be either *discarded*, or *forwarded* to the cooperative cache on a target client.

A replacement policy can use two factors in deciding whether or not to forward a block. First, a block is discarded if the replacement algorithm decides that the block is less valuable than any block in the cooperative cache. Otherwise, the block is forwarded to the target client which then replaces a block in its cache.

The second factor in deciding whether or not a block should be forwarded to the cooperative cache is *duplicate avoidance*. Since the cooperative cache is a resource used by all of the clients, the potential exists for uncoordinated client actions to result in several copies of the same block in the cooperative cache. These duplicate copies pollute the cooperative cache and reduce its hit rate. Thus a block should not be forwarded to the cooperative cache if it is

going to become a duplicate copy. In particular, if several clients have a copy of a block in their local caches, only one of the copies should be forwarded to the cooperative cache, and only if the cooperative cache does not already contain a copy.

If the client decides to forward the block, the choice of the target client becomes important in determining the effectiveness of the replacement policy. Thus, the target client should be chosen such that the forwarded block replaces a block less valuable than itself. The replaced block in the target client may be in the client's local cache or in the cooperative cache.

2.3 Server Caching

In a traditional distributed file system, the server maintains a cache of blocks that have been accessed by the clients. Although the server cache is lower in the storage hierarchy than the client caches and therefore has a lower hit rate, studies have shown that the server cache is still effective at reducing server disk traffic and improving the performance of the file system[Baker91].

The benefits of a server cache are less apparent with cooperative caching because the vast majority of local cache misses are serviced by the cooperative cache. This raises the issue of what to do with the server memory. One possibility is to use it as a part of the cooperative cache. Alternatively, the hint-based algorithm described in this paper instead uses the server memory as a *discard cache* that offsets the impact of incorrect hints.

2.4 Cache Consistency

Cache consistency is important to cooperative caching because the choice of consistency protocol affects the feasibility of using hints to perform block lookup and replacement. Managers typically enforce consistency using *tokens* which control access to a file or block. Managers use tokens in two ways: a token *grant* allows a client to access a file or block, while a token *revocation* causes a client to invalidate its copy of the file or block. Token revocation removes stale copies of a file or block from the client caches, leaving only the up-to-date version.

A *block-based* protocol maintains consistency on each individual block. The problem with a block-based protocol is that the manager must be contacted on every local cache miss to ensure consistency. As a result, there is little point in using hints to avoid contacting the manager for block lookup as described in Section 2.1.

A *file-based* protocol is similar to the block-based except that it maintains consistency on entire files rather than blocks, allowing clients to potentially handle local cache misses without contacting the manager. One drawback

is that file-based consistency does not handle concurrent write-sharing of a file by multiple clients as efficiently as block-based consistency, but this pattern of file access is rare in distributed file systems[Baker91].

3 Previous Algorithms

The original paper on cooperative caching by Dahlin et al.[Dahlin94] described a variety of different schemes for implementing cooperative caching, and settled on one called *N-chance* as providing the best performance with the lowest overhead. The authors provided a partial refinement of the algorithm in the description of the *afs* file system[Anderson95]. A subsequent paper by Feeley et al.[Feeley95] described the *Global Memory Service* (GMS) which provided better performance than *N-chance* as well as reduced overhead. This section discusses these two algorithms. A summary of the previous algorithms, as well as the hint-based algorithm we propose in the next section, is shown in Table 1.

3.1 N-chance

The *N-chance* algorithm dynamically partitions the cache of each client between blocks needed by the local client and the cooperative cache. Managers are responsible for maintaining consistency and block location information.

The replacement policy in *N-chance* uses a combination of local LRU information and duplicate avoidance to decide the best block to replace. Clients always replace the oldest block on their LRU lists. Whether the client forwards or discards the oldest block depends on the number of copies of the block. Blocks with more than one copy are discarded, while blocks with only one copy, or *singlets*, are forwarded to another client at random. If a client does not know whether or not one of its blocks is a *singlet*, it simply asks the manager.

3.2 GMS

GMS is more general than *N-chance* in that it is a distributed shared-memory system, for which cooperative caching is only one possible use. GMS is similar to *N-chance* in that it uses managers to locate blocks in the client caches. One difference is that it does not provide a consistency mechanism.

The replacement policy used by GMS uses duplicate avoidance to determine whether a block should be discarded or forwarded to the cooperative cache. A local block is forwarded to the cooperative cache if no other copies of the block exist (or in *N-chance* terminology, the block is a *singlet*). GMS differs from *N-chance* in that each copy of a block is always tagged as to whether or

not it is a *singlet*. The manager keeps track of the number of copies of each block and notifies the appropriate client when a block becomes a *singlet*.

GMS also differs from *N-chance* in that a centralized algorithm implements the choice of the target client in the replacement policy. The algorithm collects age information of blocks in the client caches from all clients at the end of dynamically controlled time intervals called *epochs*. This manager then distributes the locations of the oldest blocks in the system back to all the clients, which is then used to determine the target client in the replacement policy. The algorithm is so designed that it reasonably approximates the global LRU policy over all the client caches.

4 A Hint-based Algorithm

The previous cooperative caching algorithms rely in part on exact information, or *facts*, to manage the cache. Although facts allow these algorithms to make optimal decisions, they increase the latency of block accesses and the load on the managers. Our goal in designing a cooperative caching algorithm is to remove the reliance on centralized control of the cooperative cache. Clients should be able to access and replace blocks in the cooperative cache without involving a manager.

Reducing the dependence of clients on managers is achieved through the use of *hints*, information that only approximates the global state of the system. The decisions made by a hint-based system may not be optimal, but managing hints is less expensive than managing facts. Hints do not need to be consistent throughout the system, eliminating the need for centralized coordination of the information flow. As long as the overhead eliminated by not using facts more than offsets the effect of making mistakes, the gamble of using hints will pay off.

The remainder of this section describes the components of a hint-based algorithm. Section 4.1 describes the hint-based block lookup algorithm. Section 4.2 describes how the replacement policy decides whether or not to forward a block to the cooperative cache. Section 4.3 discusses how the replacement policy chooses the target client for forwarding a block. Section 4.4 discusses the use of the server cache to mask replacement mistakes. Finally, Section 4.5 describes the effect of the cache consistency protocol on the use of hints.

4.1 Block Lookup

When a client suffers a local cache miss a lookup must be performed on the cooperative cache to determine if and where the block is cached. The manager performs this lookup in the previous algorithms, both increasing the

Algorithm	N-chance	GMS	Hint-based
Cache Consistency	Block-based	None	File-based
Block Location	Manager-based	Manager-based	Hint-based
Replacement Policy	Random Client	Manager-based LRU	Best-guess LRU
Duplicate Avoidance	Non-singlets deleted	Non-singlets deleted	Master Copy
Server Caching	Traditional	Traditional	Discard

Table 1: **N-chance and GMS.** This table lists the key features of the N-chance, GMS, and hint-based algorithms.

block access time and incurring load on the manager and network.

An alternative approach is to let the client itself perform the lookup, using its own hints about the locations of blocks within the cooperative cache. These hints allow the client to access the cooperative cache directly, avoiding the need to contact the manager on every local cache miss.

Based on the above, we can identify two principal functions for a hint-based lookup algorithm:

- *Hint Maintenance:* The hints must be maintained so that they are reasonably accurate, otherwise the overhead of looking for blocks using incorrect hints will be prohibitive.
- *Lookup Mechanism:* Hints are used to locate a block in the cooperative cache, but the system must be able to eventually locate a copy of the block should the hints prove wrong.

Each of these functions is discussed in detail in the following sections.

4.1.1 Hint Maintenance

To make sure that hints are reasonably accurate, our strategy is to change hints only when necessary. In other words, correct hints are left untouched and incorrect hints are changed when correct information is made available.

To keep hints as correct as possible, we introduce the concept of a *master copy* of a block. The first copy of a block to be cached by any client is called the master copy. The master copy of a block is distinct from the block's other copies because the master copy is obtained from the server.

Based on this concept of the master copy, we enumerate two simple rules for hint maintenance:

1. When a client obtains a token for a file from a manager, it is also given a set of hints that contain the probable location of the master copy for each block in the file. The manager obtains the set of hints for the file from the last client to acquire a token for the file, because the last client is likely to have the most accurate hints.

2. When a client forwards a master copy of a block to a second client, both the clients now update their hints to show that the probable location of the master copy of the block is the second client.

The hints only contain the probable locations of the master copy and hence, we ignore the changes to the locations of the other copies of the block. This simplifies the task of keeping the hints accurate.

4.1.2 Lookup Mechanism

Given hints about the probable location of the master copy of a block, the lookup mechanism must ensure that a block lookup is successful, regardless of whether the hints are right or wrong. Fortunately, as all block writes go through to the server, it always has a valid copy of a block and can satisfy requests for the block should the hints prove false. This simplifies the lookup mechanism which is outlined below:

1. When a client has a local cache miss for a block, it consults its hint information for the block.
2. If the hint information contains the probable location for the master copy of the block, the client forwards its request to this location. Otherwise, the request is sent to the server.
3. The client which receives a forwarded request for a block consults its hint information for the block and proceeds to Step 2.

The general idea is that each client keeps track of the probable location of the master copy of each block, and uses this information to lookup blocks in the cooperative cache.

Simulations show this algorithm works well in a distributed UNIX environment, as described in Section 6. However, the algorithm will not be effective in the following scenario. If several clients share a working set of blocks larger than the cooperative cache, the locations of the master copies will change rapidly as blocks move in and out of the client caches. This will cause the probable master copy locations to be inaccurate, leading to excessive forwarding of requests.

4.2 Forwarding

When a block is ejected from the local cache of a client, the cooperative cache replacement policy decides whether or not the block should be forwarded to the cooperative cache. As discussed earlier, one of the motivations of the replacement policy is to ensure that only one copy of a block is stored in the cooperative cache. If not, the cooperative cache will contain unnecessary duplicate copies of the same block.

The previous algorithms rely on the manager to determine whether or not a block should be forwarded to the cooperative cache. A block is forwarded if it is the only copy of the block stored in either the local caches or the cooperative cache. Maintaining this invariant is expensive, however, requiring an N-chance client to contact the manager whenever it wishes to discard a block that is not known to be a singlet, and the GMS manager to contact a client whenever a block becomes a singlet.

To avoid these overheads, we propose a forwarding mechanism in which the copy to be forwarded to the cooperative cache is predetermined and does not require communication between the clients and the manager. In particular, only the master copy of a block is forwarded to the cooperative cache, while all other copies are discarded. Since only master copies are forwarded, and each block has only one master copy, there can be at most one copy of a block in the cooperative cache.

A potential drawback of the master copy algorithm is that it has a different forwarding behavior than the previous algorithms. Instead of forwarding the last local copy of a block as in GMS or N-chance, the master copy algorithm forwards the first or master copy. In some cases, this may lead to unnecessary forwardings. A block which is deleted before it is down to its last copy should not have been forwarded to the cooperative cache. The existing algorithms avoid this, while the master copy algorithm will potentially forward the block. Fortunately, our measurements show that few of the master copy forwardings are unnecessary, as described in Section 6.3.

4.3 Best-Guess Replacement

Once the replacement policy has decided to forward a block to the cooperative cache, it must decide the target client of this forwarding. Forwarding a block to this target client will replace a block that is either in the client's local cache or in the cooperative cache. Note that this replaced block can be a master copy, providing a means for removing master copies from the cooperative cache.

Previous algorithms either choose the client at random, or rely on information from the manager to select the target. An alternative based on hints, however, can provide highly accurate replacements without requiring a

centralized manager. We refer to this as *best-guess* replacement because each client chooses a target client that it believes has the system's oldest block. The objective is to approximate global LRU, without requiring a centralized manager or excessive communication between the clients. The challenge is that the block age information is distributed among all the clients, making it expensive to determine the current globally LRU block.

In best-guess replacement, each client maintains an *oldest block list* that contains what the client believes to be the oldest block on each client along with its age. This list is sorted by age. A block is forwarded to the client that has the oldest block in the oldest block list.

The high accuracy of best-guess replacement comes from exchanging information about the status of each client. When a block is forwarded from one client to another, both clients exchange the age of their current oldest block, allowing each client to update its oldest block list. The exchange of block age information during replacement allows both *active* clients (clients that are accessing the cooperative cache) and *idle* clients (clients that are not) to maintain accurate oldest block lists. Active clients have accurate lists because they frequently forward blocks. Idle clients will be the targets of the forwardings, keeping their lists up-to-date as well. Active clients will also tend to have young blocks, preventing other clients from forwarding blocks to them. In contrast, idle clients will tend to accumulate old blocks and therefore be the target of most forwarding requests.

Changes in the behavior of a client may cause the oldest block lists to become temporarily inaccurate. An active client that becomes idle will initially not be forwarded blocks, but its oldest block will age relative to the other blocks in the system. Eventually this block will be the oldest in the oldest block lists on other clients and be used for replacement. On the other hand, an idle client that becomes active will initially have an up-to-date list because of the blocks it was forwarded while idle. This allows it to accurately forward blocks. Other clients may erroneously forward blocks to the newly-active client but once they do, their updated oldest block lists will prevent them from making the same mistake twice.

Although trace-driven simulation has shown this simple algorithm to work well, there are several potential problems, including the effect of replacing a block that is not the globally LRU block and also the problem of overloading a client with simultaneous replacements.

First, since global state information is not maintained, it is possible for a client to replace a block that is not the globally LRU block. However, if the replaced block is *close* to the globally LRU block, the performance impact of not choosing the globally LRU block is minimal. In addition, the next section discusses a mechanism for

masking any deviations from the globally LRU block.

Second, if several clients believe that the same client has the oldest block, they will all forward their blocks to that client, potentially overloading it. Fortunately, it can be shown that it is highly unlikely that the clients using the cooperative cache would forward their blocks to the same target. This is because clients that do forward their blocks to the same target will receive different ages for the oldest block on the target, since each forwarded block replaces a different oldest block. Over time, the clients' oldest block lists will contain different block age information even if they start out identical, reducing the probability of always choosing the same forwarding targets.

4.4 Discard Cache

Type of block	Action
Non-master copy	Discard
Old master copy	Discard
Young master copy	Send to discard cache

Table 2: **Discard Cache Policy.** This table lists how the hint-based replacement policy decides which blocks to send to the discard cache. A master copy is *old* if it is older than all blocks in the oldest block list, else it is considered *young*. The oldest block list is the per-client list that contains what the client believes to be the oldest block on each client along with its age.

One drawback of best-guess replacement is that erroneous replacements will occur. A block may be forwarded to a client that does not have the oldest block; indeed, a block may be forwarded to a client whose oldest block is actually younger than the forwarded block.

To offset these mistakes we introduce the notion of a *discard cache*, one that is used to hold possible replacement mistakes and thus increase the overall cache hit rate of the system. The simple algorithm used to determine whether a block is mistakenly replaced and should be sent to the discard cache is shown in Table 2. As is evident, non-master copies are always discarded because only master copies are accessed in the cooperative cache.

Replacements are considered to be in error when the target client of a replacement decides that the block is too young to be replaced. A client chooses to replace a block on a particular target client because it believes that client contains the oldest block. The target client considers the replacement to be in error if it does not agree with this assessment. The target determines this by comparing the replaced block's age with the ages of the blocks on its oldest block list. If the block is younger than any of the blocks on the list, then the replacement is deemed an error and the block is forwarded to the discard cache. Otherwise, the block is discarded.

The blocks in the discard cache are replaced in global LRU order. Thus the discard cache serves as a buffer to hold potential replacement mistakes. This extends the lifetimes of the blocks and reduces the number of erroneous replacements that result in an expensive disk access.

4.5 Cache Consistency

The use of hints for block lookup raises the issue of maintaining cache consistency. One solution is to use block-based consistency, but this would require contacting the manager on every local cache miss to locate an up-to-date copy, making it pointless to use hints for block lookup or replacement. For this reason, we propose the use of a file-based consistency mechanism. Clients must acquire a token from the manager prior to accessing a file. The manager controls the file tokens, revoking them as necessary to ensure consistency. The token includes version numbers for all the file's blocks, allowing copies of the blocks to be validated individually. Once a client has the file's token it may access the file's blocks without involving the manager, enabling the use of hints to locate and replace blocks in the cooperative cache.

5 Methodology

This section describes the methodology used to compare the performance of the N-chance, GMS, hint-based, and ideal algorithms in detail and analyze the design tradeoffs. We describe the simulation environment, the criteria for evaluating the algorithms, and the ideal algorithms against which the other algorithms are compared.

5.1 Simulation Environment

The algorithms were evaluated using trace-driven simulation. The traces of the Sprite distributed file system [Baker91] were used to drive the simulator. These traces cover four two-day *periods*, and record file system accesses by user programs, such as opening and closing files, and seeking on file descriptors. Actual read and write events were not recorded, but can be inferred from file offsets in other records. The traces record application-level behavior and thus are not Sprite-specific. We restricted our use of the traces to the main file server *allspice*. Table 3 shows statistics for each of the trace periods, while Table 4 shows the simulation parameters.

Most of the simulation parameters are derived from the original study on cooperative caching by Dahlin et al.[6], simplifying performance comparisons. The various access times were obtained from previously published measurements. Although these measurements are now a few years old, and thus likely to be slow when compared to

Trace Parameter	Period			
	1	2	3	4
Block reads	276,628	2,011,915	261,023	343,189
Unique blocks accessed	53,349	13,108	33,063	75,273
Active clients	32	24	38	34

Table 3: **Trace Period Statistics.** This table contains statistics the four trace periods. *Active clients* refers to the number of clients that actually used the cooperative at any point during the period.

Clients	42
Servers	1
Managers	1
Client Cache Size	16 MB
Server Cache Size	128 MB
Block Size	8 KB
Local Memory Access Time	0.25 ms
Remote Memory Access Time	1.25 ms
Disk Access Time	15.85 ms
Write policy	write-through
Warm-up Block Accesses	400,000
Forwarding Cache Entries	100
Network	ATM
Message Latency	0.2 ms

Table 4: **Simulation Parameters.** This table describes the environment used to evaluate the various cooperative caching algorithms.

state-of-the-art equipment, they were obtained from real systems. Furthermore, we did not believe that the benefit of updating the simulation parameters warranted the increased difficulty in comparing results.

We also assume that there is a single manager handling centralized functions such as cache consistency and block location. This makes it easier to measure the manager load imposed by the different systems, without introducing an algorithm to distribute the load over multiple managers.

The N-chance and GMS simulators used in these simulations were derived from the simulators created by the systems' designers. The N-chance simulator was modified to incorporate additional functionality used in the *xf*s file system[Anderson95]. In the modified system, a manager preferentially forwards a request to the cooperative cache instead of a server, improving the cooperative cache hit rate and reducing the load on the servers.

The GMS simulator was modified to add a file-based consistency mechanism. The original GMS system did not contain a consistency mechanism, making it difficult to use for cooperative caching. The consistency mechanism we implemented is identical to the one we use in our hint-based algorithm, as described in Section 4.5.

The simulators were validated through both extensive debugging and analysis of their behavior. For all simulators, we analyzed in detail their processing of a small

sample of the traces. Also, all simulators produce expected results for the traces if cooperative caching was disabled. The GMS simulator was further validated by running the traces while varying the length of the epochs. As the epochs were lengthened, the average age of the replaced blocks decreased as expected. Finally, the hint-based simulator was validated both by inspection of its processing of a sample of the traces, and by comparing its results with those of the N-chance simulator.

5.2 Evaluation Criteria

We use the following two metrics to evaluate the performance of the cooperative caching algorithms:

- **Average Block Access Time:** This metric measures the average time required to access a block. The access time is determined by the hit rates in the different layers of the storage hierarchy. Algorithms that make better use of the local and cooperative caches to avoid disk accesses will have low access times. Access time is only measured for block reads because all algorithms use write-through caches.
- **Overhead:** This metric measures the work required to implement cooperative caching. This overhead is broken down into load on the managers, and network messages required to satisfy a local cache miss and replace a block in the cooperative cache.

5.3 Ideal Algorithms

For comparison purposes, the performance of two ideal cooperative caching algorithms is included. These algorithms provide an upper bound on cooperative cache performance, and thus provide an absolute yardstick against which other algorithms may be measured. The ideal algorithms differ in the replacement policy they use. The Optimal algorithm replaces blocks in a fashion that maximizes the hit rate on the cooperative cache, but cannot be implemented because it requires predicting the future. The Global LRU algorithm approximates the Optimal algorithm by replacing the oldest block in the system, but cannot be efficiently implemented because of the overhead required to find and replace the oldest block.

5.3.1 Optimal

The Optimal replacement always replaces the block whose next access is farthest in the future. It has been shown that this replacement policy is optimal because it minimizes the number of cache misses [Belady66] and therefore has the lowest block access time.

The Optimal replacement algorithm for a cooperative cache differs from that in a virtual memory system in that block location must be considered when determining which block to replace. When a client stores a block to the cooperative cache, there may be several candidate blocks to replace which are never accessed in the future. To reduce the number of block transfers, the Optimal algorithm always replaces blocks stored locally at that client, if possible.

5.3.2 Global LRU

The Global LRU algorithm approximates the Optimal algorithm by collecting information about the LRU block of every client to determine the globally LRU block and then replacing it. In a real implementation, this algorithm would not only be expensive but also inaccurate. The inaccuracy stems from the fact that the LRU block of a client may change while another client is collecting information about the LRU blocks of all clients.

The Global LRU replacement policy is based on the two goals of determining the exact globally LRU block and minimizing the number of copies of a block. When a client needs to forward a block to the cooperative cache, it first checks to see if the block is a duplicate. If so, it is discarded, otherwise it is forwarded to the client storing the globally LRU block and the globally LRU block is discarded.

As a result of its replacement policy, the Global LRU algorithm tries to maximize the local and remote cache hit rates but does not try to minimize the number of block transfers. The algorithm always discards the globally LRU block, usually requiring a block transfer. As a result, an algorithm that approximates Global LRU and attempts to minimize block transfers can have a lower replacement overhead than the Global LRU algorithm.

6 Performance

This section describes the performance of the cooperative caching algorithms when simulated using the Sprite traces. They are compared in terms of average block access time, manager load, lookup messages, and replacement messages. The effectiveness of the discard cache is also measured, as is the sensitivity of the block access time to variations in the simulation parameters.

6.1 Block Access Time

Figure 1 shows the average block read access time for all the algorithms, broken down by the time spent in accessing the local cache, remote client caches, server cache, and server disk. Write accesses are not included in the figure because all algorithms use write-through caches.

The read access times for the GMS and hint-based algorithm are very close to the ideal algorithms, and they spent similar amounts of time handling hits in the different levels of the storage hierarchy. The performance of the N-chance algorithm is similar to that of the optimal algorithms in all but the second and fourth periods. During these periods, the N-chance algorithm has a higher number of disk accesses and remote client hits, leading to a longer access time. This is caused by more block accesses in these periods than in the other periods. This increase affects N-chance more than the other algorithms because of its random replacement policy. The higher number of blocks accessed increases the probability that a block replaced at random was being used by the client that cached it.

The time breakdown shows that hits in the server cache are nearly non-existent for all algorithms except the hint-based. This indicates that the hint-based algorithm occasionally makes mistakes in replacing a block from the cooperative cache, but the discard cache in the server corrects some of them.

The block access times for the GMS and hint-based algorithms are very close to that of the ideal algorithms, leaving little room for improvement. The performance of the hint-based algorithm is particularly encouraging, given that hints can be occasionally incorrect.

The block location hints for the cooperative cache are highly accurate. For only 0.01% of the local cache misses (averaged across all periods) is the desired block in the cooperative cache but the hints say it is not. In these cases, the client will erroneously retrieve the block from the server. Conversely, when a hint says a block is in the cooperative cache, it is correct for 99.95% of all local cache misses. Of these correct hints, 97.56% point to the actual location of the block while the remaining result in requests being forwarded. The high hint accuracy and the small number of forwardings translate into an average of only 2.03 messages to perform a block lookup. In comparison, both N-chance and GMS require 3 messages per lookup.

6.2 Manager Load

The load imposed on the manager is one measure of the overhead and scalability of an algorithm. The less work a centralized manager must do on the client's behalf, the more scalable is the system. Figure 2 shows the manager

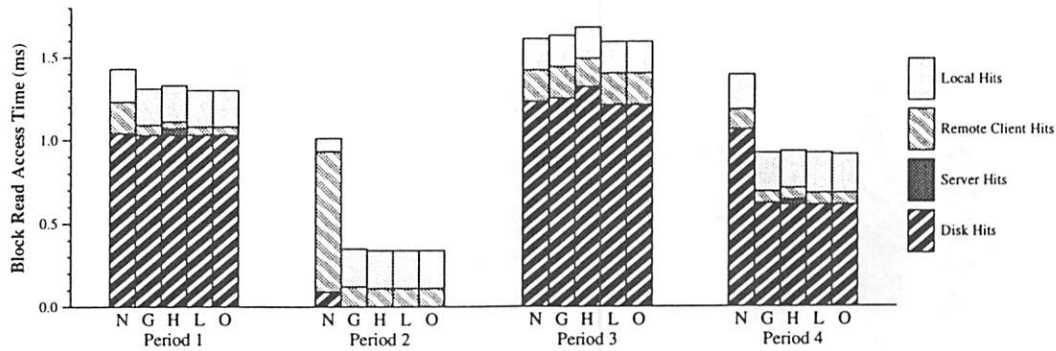


Figure 1: **Block Access Time.** This figure shows the average block read access times for the N-chance(N), GMS(G), hint-based(H), Global LRU(L) and Optimal(O) algorithms for each period of the Sprite traces. The segments of the bars show the fraction of the block read access time contributed by hits to the local cache, cooperative cache, server cache, and server disk.

load categorized by the load's source, and expressed as the number of messages sent and received by the manager. This is a valid measure of the manager load because each message sent and received by the manager represents work performed by the manager.

As can be seen, managing the client cache consistency imposes a very small load on the manager. This does not mean that the choice of consistency algorithm does not affect system performance, only that it does not contribute significantly to manager load. File-based consistency is still important for enabling the use of hints for replacement and lookup.

Replacement and lookup traffic account for nearly all of the manager load for the N-chance and GMS algorithms. The clients must contact the manager each time a block is forwarded and each time a lookup is done, whereas the hint-based algorithm allows the clients to perform these functions themselves. The result is that the manager load is much higher for N-chance and GMS. N-chance has a relatively high manager load during the second and fourth periods because it has a low local cache hit rate in those periods, as described in the previous section. This increases the number of blocks that must be accessed from the cooperative cache or the server, and increases the manager load accordingly.

6.3 Best-Guess Replacement

Figure 2 showed that at about half of the N-chance and GMS manager load is due to managing block replacement. In this section we analyze the overheads associated with block replacement.

Figure 3 depicts the number of replacement messages each algorithm requires to handle a local cache miss. The N-chance and GMS algorithms have three sources of replacement messages: forwarding the block to another client and notifying the manager; notifying the manager

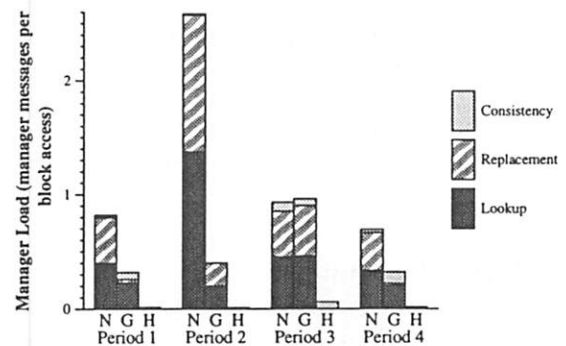


Figure 2: **Manager Load.** This shows the average manager load of the N-chance(N), GMS(G) and hint-based(H) algorithms in the Sprite traces. The manager load is defined as the number of messages received and sent by the manager per block access. The load is categorized by its cause: consistency, replacement and lookup.

when a block is deleted; and exchanging messages between the clients and the manager to determine when a block should be discarded as opposed to forwarded. Except for the actual forwarding of the block to another client, all messages involve the manager, increasing its load. For best-guess replacement, the only message required is the one to forward the master copy of a block to another client. This dramatically reduces the total number of replacement messages required per local cache miss.

As mentioned before, one of the potential drawbacks of the master copy algorithm is that it may unnecessarily forward master copies. Although Figure 3 shows that best-guess replacement outperforms the other algorithms despite this drawback, we also measured the fraction of forwardings that were unnecessary. An average of only 2.11% were unnecessary across all periods, indicating that they have a negligible effect on performance.

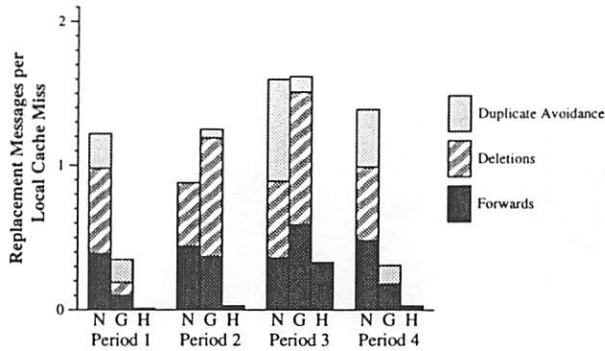


Figure 3: Replacement Traffic. This figure shows the number of replacement messages required per local cache miss in the N-chance(N), GMS(G) and hint-based algorithms(H). The messages are categorized by those required to forward a block, delete a block, and keep track of the number of copies of a block (duplicate avoidance). For N-chance and GMS, this includes two messages per singlet forwarded (one to forward the block and another to notify the manager), one message per block deleted, as well as the messages required to keep track of singlets. For the hint-based algorithm, this includes one message per master copy forwarded.

6.4 Discard Cache

Server Memory	Hit Ratio	Block Access Time(ms)
Disk Cache	0.35%	1.01
Cooperative Cache	1.65%	0.81
Discard Cache	2.16%	0.74

Table 5: Server Memory Uses. This table shows how various uses of the server memory affect the block access time of the hint-based system. Server memory is used as either a traditional disk cache, as part of the cooperative cache, or as a discard cache. The results are averaged across all periods of the trace.

The server memory represents a valuable resource to the system and at 128 MB it constitutes a large fraction of the memory of the system. The hint-based system we propose uses the server memory as a discard cache to mask some of the mistakes made by the best-guess replacement policy. There are other possible uses for the server memory, however, including as a traditional disk cache and as a portion of the cooperative cache. Unfortunately, the default 16 MB client cache size used in the simulations makes it difficult to measure the effectiveness of the discard cache. The cooperative cache is so large and effective that few accesses go to the server. Thus, to measure the effectiveness of the discard cache, we reduced the size of the client caches and server cache to 4 MB and 16 MB, respectively, and ran simulations of the hint-based system with the server memory in the different uses mentioned. Reducing the cache sizes increases the miss rates on the

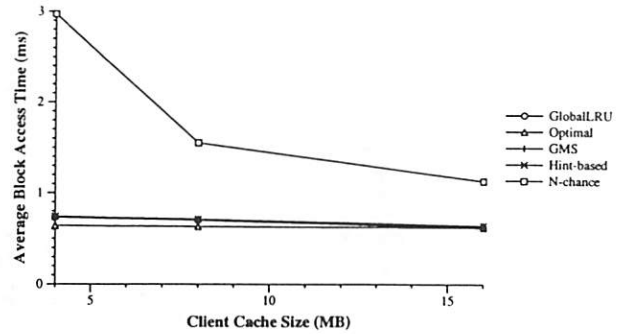


Figure 4: Access Time vs. Cache Size. This figure shows the average block read access time for the algorithms as a function of the client cache size.

local and cooperative caches, and therefore the load on the server memory.

The results are shown in Table 5 and indicate that when the server memory is used as a traditional disk cache, it has a very low hit rate of 0.35% because most of the blocks it stores are duplicated in the local and cooperative caches. This results in a block access time of 1.01 ms. If the server memory is instead used as part of the cooperative cache, the hit rate increases by nearly a factor of 5, causing the block access time to drop to 0.81 ms. Using the memory as a discard cache, however, further increases the hit rate to 2.16% and drops the block access time to 0.74 ms. By masking some of the replacement mistakes, the discard cache provides a 9% improvement in the block access time over using the memory as part of the cooperative cache.

6.5 Sensitivity

The analysis presented in the previous sections was based on a single system configuration, in which the number of clients, client cache size, number of servers, and other parameters were fixed. Although the hint-based algorithm performed well under the chosen configuration, its sensitivity to variations in the environment is also of concern. This section presents the sensitivity of the block access time to two environmental variables: the client cache size and the fraction of the clients that actively use the cooperative cache.

First, Figure 4 shows the block access time as the client cache size is varied from 4 MB to 16 MB. The remaining system parameters are the same as those shown in Table 4. A smaller client cache increases the load on the cooperative cache in two ways: first, it increases the local cache miss rates and therefore accesses to the cooperative cache; and second, it reduces the available size of the cooperative cache. As the figure shows, the block access times for most algorithms does not significantly increase with a

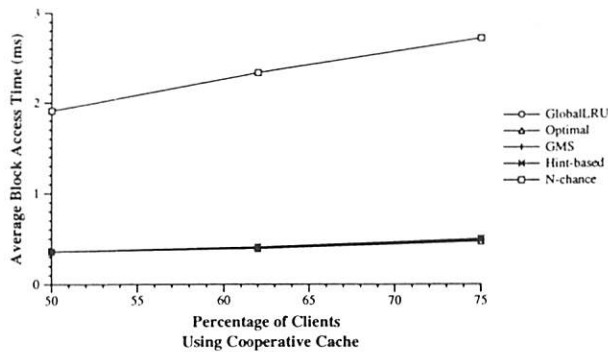


Figure 5: **Access Time vs. Active Clients.** This figure shows the average block read access time for the algorithms as a function of the fraction of clients that are using the cooperative cache during the period. The fraction of clients using the cooperative cache was varied by removing idle client trace records from the trace. Due to the difficulty in doing this without affecting the workload behavior, only the second period was used.

4 MB client cache, although they do slightly diverge from optimal. Even with caches this small the algorithms do a good job of finding and using the available idle memory, producing access times that are close to optimal. The exception is the N-chance algorithm, whose policy of randomly forwarding blocks hurts it when cooperative cache is scarce.

The sensitivity of the block access time to the fraction of clients that are using the cooperative cache is also of interest. Increasing the fraction of clients that use the cooperative cache increases the demand on the cache, and also decreases the cooperative cache size. This combined effect increases the importance of managing the cooperative cache efficiently.

As Figure 5 shows, only the performance of the N-chance algorithm declines as the fraction of clients using the cooperative cache increases. Again, this is due to the random forwarding of blocks to other clients in N-chance. The remaining algorithms all perform close to the optimal.

7 Related Work

Cooperative caching for file systems developed from research involving remote memory usage. The idea of remote memory servers in distributed systems was first introduced by Comer and Griffioen in [Comer90]. Felten and Zahorjan proposed the use of idle machines as remote memory servers in [Felten91]. Franklin et al. in [Franklin92] introduced the concept of remote client servers to extend the traditional client-server database architecture. Leff et al. in [Leff91] showed that memory must be dynamically partitioned between local and remote

client needs to maximize hit rates.

Our use of hints to perform block lookup is similar to the techniques used to perform page lookup in distributed shared memory systems that support parallel computation. Li and Hudak describe several strategies for managing distributed shared pages [Li89], including a dynamic distributed manager algorithm in which nodes send page requests to the probable owner of the page. If the target node does not have the page, it forwards the request to the node it believes to be the probable owner. Unlike the hint-based algorithm we propose, all nodes keep track of probable owner information for all pages, so that the request eventually reaches the correct owner. Their results show that the probable owner information is quite accurate and the actual number of forwardings is very small. Our hint-based algorithm also differs in that blocks can be forwarded to the cooperative cache, necessitating a distributed replacement policy. The work of Li and Hudak relies on the virtual memory systems of the individual machines to swap pages to disk, rather than forwarding pages to other nodes.

Cooperative caching is also related to multiprocessor caching in shared memory machines [Lenoski90]. However, message costs are greater in distributed file systems than in multiprocessors and have a greater impact on performance. Thus there must be a concerted effort to reduce the number and size of messages required. This is one focus of research on distributed shared memory [Carter91].

The discard cache is similar in purpose to the victim cache proposed by Jouppi [Jouppi90]. A victim cache is a small fully-associative miss cache that is placed between a direct-mapped processor cache and the main memory system. The victim cache is loaded with the victim of a cache miss rather than the missed cache line itself. As a result, cache lines that conflict in the processor cache can both be cached in the victim cache, increasing performance. In essence, the victim cache catches replacement mistakes made by the processor cache because it is direct-mapped. The discard cache, in contrast, catches mistakes made because of incomplete information about block ages.

8 Conclusions

Cooperative caching is a technique that allows clients to access blocks stored in the memory of other clients. This enables some of the local cache misses to be handled by other clients, offloading the server and improving the performance of the system. However, cooperative caching requires some level of coordination between the clients to maximize the overall system performance. Previous cooperative caching algorithms achieved this coordination by maintaining global information about the system state. This paper shows that allowing clients to make local de-

cisions based on hints performs as well as the previous algorithms, while requiring less overhead. The hint-based algorithm's block access times are as good as those of the previous and ideal algorithms, while reducing manager load by more than a factor of 15, block lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

9 Acknowledgments

We would like to thank Mike Dahlin for providing the N-chance simulator and the Sprite trace information, and Mike Feeley for clarifications on the GMS system. Wanda Chiu, Matti Hiltunen, Mudita Jain, Dave Lowenthal, Anup Kuzhiyil and Todd Proebsting all provided much-needed comments on early drafts of this paper, as did the anonymous reviewers. Our paper shepherd, Peter Chen, also deserves thanks for his efforts to improve the paper. This work has been funded in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-94-C-0049 and DABT63-95-C-0075, and a grant from Intel Corporation.

References

- [Anderson95] T. E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 109–126, December 1995.
- [Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212, October 1991.
- [Belady66] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [Carter91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 152–164, October 1991.
- [Comer90] Douglas E. Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the Summer 1990 Usenix Conference*, pages 127–135, June 1990.
- [Dahlin94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to improve File System Performance. In *Proceedings of the 1st Symposium on Operating System Design and Implementation*, pages 267–280, November 1994.
- [Feeley95] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 201–212, December 1995.
- [Felten91] Edward W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, University of Washington, March 1991.
- [Franklin92] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global Memory Management in a Client-Server DBMS Architectures. In *Proceedings of the 18th VLDB Conference*, pages 596–609, August 1992.
- [Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.
- [Jouppi90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.
- [Leff91] Avraham Leff, Philip S. Yu, and Joel L. Wolf. Policies for Efficient Memory Utilization in a Remote Caching Architecture. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 198–207, December 1991.
- [Lenoski90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [Li89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions of Computer Systems*, 7(4):321–359, November 1989.
- [Nelson93] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions of Computer Systems*, 11(2):228–239, February 1993.
- [Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

Online Data-Race Detection via Coherency Guarantees

Dejan Perković and Peter J. Keleher
Department of Computer Science
University of Maryland
College Park, MD 20742-3255
(keleher|dejanp)@cs.umd.edu

Abstract

We present the design and evaluation of an on-the-fly data-race-detection technique that handles applications written for the lazy release consistent (LRC) shared memory model. We require no explicit association between synchronization and shared memory. Hence, shared accesses have to be tracked and compared at the minimum granularity of data accesses, which is typically a single word.

The novel aspect of this system is that we are able to leverage information used to support the underlying memory abstraction to perform on-the-fly data-race detection, without compiler support. Our system consists of a minimally modified version of the CVM distributed shared memory system, and instrumentation code inserted by the ATOM code re-writer.

We present an experimental evaluation of our technique by using our system to look for data races in four unaltered programs. Our system correctly found read-write data races in a program that allows unsynchronized read access to a global tour bound, and a write-write race in a program from a standard benchmark suite. Overall, our mechanism reduced program performance by approximately a factor of two.

1 Introduction

While potentially very useful, data-race detection mechanisms have yet to become widespread. Part of the problem is surely the restricted domain in which most such mechanisms operate, i.e. parallelizing compilers. Such restrictions are deemed necessary because data-race detection is generally NP-complete [17], and exponential searches over a domain the size of the number of shared accesses in a program execution are prohibitively expensive.

This paper presents the design and evaluation of an online data-race detection technique for explicitly parallel shared-memory applications. This technique is applicable

for shared memory programs written for the lazy-release-consistent (LRC) [9] memory model. Our work differs from previous work [3, 4, 5, 7, 16, 15] in that data-race detection is performed both on-the-fly and without compiler support. In common with other dynamic systems, we address only the problem of detecting data races that occur in a given execution, not the more general problem of detecting all races allowed by program semantics [17].

Our general approach is to run applications on a modified version of the Coherent Virtual Memory (CVM) [11, 12] system, a distributed shared memory (DSM) system that supports LRC. DSMs support the abstraction of shared memory for parallel applications running on CPUs connected by general-purpose interconnects, such as networks of workstations or distributed memory machines like the IBM SP-2. The key intuition of this work is the following:

LRC implementations already maintain enough ordering information to make a constant-time determination of whether any two accesses are concurrent.

Hence, a DSM that implements LRC can perform the entire process on-the-fly with acceptable overhead.

Modifying CVM to implement data-race detection consisted of (i) adding instrumentation to detect read accesses, (ii) integrating this information into existing CVM structures that already contain analogous information about write accesses, and (iii) running a simple race-detection algorithm at existing global synchronization points. The task of this last point is made much easier by leveraging off of ordering information already maintained to support consistency guarantees.

We used this technique to check for data races in implementations of four common parallel applications. Our system correctly found races in two. TSP, a program that solves the Traveling Salesman Problem, has a large number of data races that result from unsynchronized read accesses to a global tour bound. The reads are left unsynchronized to improve performance; out-of-date tour bounds may cause

redundant work to be performed, but do not violate correctness. Water-Nsquared, of the Splash2 [22] benchmark suite, had a data race that constituted a real bug. This bug has been reported to the Splash authors and fixed in their current version.

While overhead is still potentially exponential, we describe a variety of techniques that greatly reduce the number of comparisons that have to be made. Those portions of the race-detection procedure that have the largest theoretical complexity are only the third or fourth-most expensive portion of the overall technique for the applications that we studied. Specifically, we show that i) we can statically eliminate over 99% of all load and store instructions as potential race participants, ii) we dynamically eliminate over 70% of all program execution from consideration by using LRC ordering information, and iii) the slowdown from using data-race detection in our system is approximately a factor of two for the applications studied. While this overhead is clearly too high for the system to be used all of the time, it is low enough for use when traditional debugging techniques are insufficient.

2 Problem Definition

The goal of this work is to create a system that detects race conditions online. Since our strategy relies on LRC consistency, our system is clearly applicable only for applications that will run properly on release-consistent systems, i.e. properly-labeled [6] or DRF1 [1] applications. The following definitions are assumed throughout the rest of the paper.

Definition 1 A data race is defined as a pair of memory accesses in some execution, such that:

1. Both access the same shared variable,
2. At least one is a write,
3. The accesses are not ordered by system-visible synchronization or program order.

In the sense discussed by Netzer [18], the races found by our system are *actual* data races, i.e. they are races that occur while the program is running on our system. In common with most other implemented systems, both with and without compiler support, we make no claim to detect all *feasible* data races, i.e. all data races allowed by the semantics of the program. As such, a program running to completion on our system without data races is not a guarantee that subsequent executions will be free of data races as well. In practice, however, we expect most data races to reveal themselves when given an appropriate input set.

Figure 1 shows a portion of a single execution in which two processes access shared variable x , and synchronize through synchronization variable L . Both w_1-r_1 and w_1-r_2

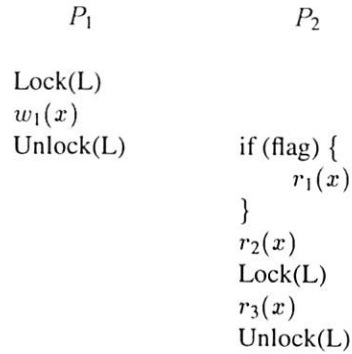


Figure 1. Race Conditions

are *feasible* data races, assuming we have no knowledge of the value of `flag`. However, if `flag` is equal to zero during an execution, w_1-r_2 is the only *actual* data race, the only data race that occurs and would therefore be caught by our system. The access pair w_1-r_2 is still potentially a bug for some other execution, but would not be flagged by CVM during this execution. Shared accesses w_1 and r_3 do not constitute a data race, as they are ordered by P_1 's unlock and P_2 's lock.

In order for our system to distinguish between w_1-r_1 and w_1-r_3 in Figure 1, the system must be able to detect and understand the semantics of all synchronization used by the programs. In practice, this requirement means that programs must use only system-provided synchronization. Any synchronization implemented on top of the shared memory abstraction is invisible to the system, and could result in spurious race warnings.

However, the above requirement is no stricter than that of the underlying DSM system. Programs must use system-visible synchronization in order to run on any release-consistent system. Our data-race detection system imposes no additional consistency or synchronization constraints.

Given the above definition for data races, our system will detect all data races that occur during a given execution.

3 Lazy Release Consistency and Data Races

3.1 Lazy Release Consistency

Lazy release consistency [9] is a variant of *eager* release consistency (ERC) [6], a relaxed memory consistency that allows the effects of shared memory accesses to be delayed until selected synchronization accesses occur. Simplifying matters somewhat, shared memory accesses are labeled either as *ordinary* or as *synchronization* accesses, with the

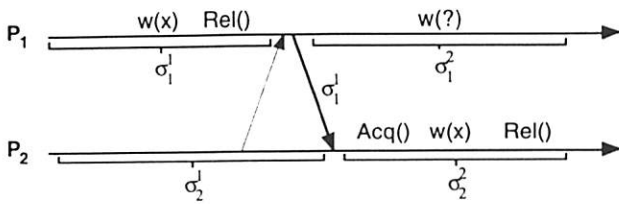


Figure 2. Process Intervals

latter category further divided into *acquire* and *release* accesses. Acquires and releases may be thought of as conventional synchronization operations on a lock, but other synchronization mechanisms can be mapped on to this model as well. Essentially, ERC requires ordinary shared memory accesses to be performed only when a subsequent release by the same processor is performed. ERC implementations can delay the effects of shared memory accesses as long as they meet this constraint.

Under LRC protocols, processors further delay performing modifications remotely until subsequent acquires by other processors, *and* the modifications are only performed at the other processor that performed the acquire. The central intuition of LRC is that competing accesses to shared locations in correct programs will be separated by synchronization. By deferring coherence operations until synchronization is acquired, consistency information can be piggy-backed on existing synchronization messages.

To do so, LRC divides the execution of each process into *intervals*, each identified by an *interval index*. For example, Figure 2 shows an execution of two processors, each of which have two intervals. Interval 1 of P_1 , denoted σ_1^1 , contains a release synchronization access and a write to shared variable x . Each time a process executes a release or an acquire, a new interval begins and the current interval index is incremented. Intervals of different processes are related by a *happens-before-1* partial ordering [1]:

1. intervals on a single processor are totally ordered by program order,
2. interval σ_p^i precedes interval σ_q^j if σ_q^j begins with the acquire corresponding to the release that concluded interval σ_p^i , and
3. the transitive closure of the above.

LRC protocols append consistency information to all synchronization messages. This information consists of structures describing intervals seen by the releaser but not the acquirer. For example, the message granting the lock to P_2 in Figure 2 contains information about all intervals seen by P_1 at the time of the release that had not yet been seen by P_2 , i.e. σ_1^1 .

3.2 Data Race Detection in an LRC System

The *happens-before-1* relation orders intervals, and by implication, accesses within intervals. Since *happens-before-1* is a combination of synchronization order (the release by P_1 precedes the acquire by P_2), and program order, it is clear that the write to x in σ_1^1 of Figure 2 precedes (via the *happens-before-1* relation) the write in σ_2^2 (interval 2 of P_2).

We can now re-define Definition 1 as follows:

Definition 2 A data race is defined as a pair of memory accesses in some execution, such that:

1. Both access the same shared variable,
2. At least one is a write,
3. The accesses are not ordered with respect to happens-before-1.

More informally, a data race is a pair of accesses that do not have intervening synchronization, such that at least one of the accesses is a write. In Figure 2, if the second write of P_1 were to variable x , it would constitute a data race with the access in σ_2^2 , because intervals σ_1^2 and σ_2^2 are concurrent (not ordered).

In general, detecting data races requires comparing each access against every other access. With an LRC system, however, we can limit comparisons only to accesses in pairs of concurrent intervals. For example, interval pair σ_1^1 - σ_2^2 in Figure 2 is not concurrent (among others), and so we do not have to check further in order to determine if there is a data race formed by accesses of these intervals. Furthermore, for each concurrent interval pair, we only perform word-level comparisons if we have first verified that the pages accessed by the two intervals overlap.

For example, assume that the second write by P_1 in Figure 2 is to a variable y that is located on the same page as x . A comparison of pages accessed by concurrent intervals σ_1^2 and σ_2^2 would reveal that they access overlapping pages, and hence we would need to perform a bitmap comparison in order to determine if the accesses constitute false sharing or true sharing (i.e. a data race). In this case, the answer would be false sharing because the accesses are to different locations. However, if P_1 's second write were to z , a variable on a completely different page, our comparison of pages accessed by the two intervals would reveal no overlap. No bitmap comparison would be performed, even though the intervals are concurrent.

4 Implementation

We implemented our data-race detection on top of CVM [11, 12], a software DSM that supports multiple protocols and consistency models. Like commercially available

	Input Set	Synchronization	Memory Size (kbytes)	Intervals Per Barrier	Slowdown (8 Proc)
FFT	64 x 64 x 16	barrier	3088	2	2.08
SOR	512x512	barrier	8208	2	1.83
TSP	19 cities	lock	792	177	2.51
Water	216 mols, 5 iters	lock, barrier	152	46	2.31

Table 1. Application Characteristics

systems such as TreadMarks [10], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, lightweight threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base *Page* and *Protocol* classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events take place. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized system, TreadMarks, running a similar protocol. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base. In order to simplify the comparison process, however, we do not use either of these techniques in this study.

We made only three modifications to the basic CVM implementation: (i) we added instrumentation to collect read and write access information, (ii) we added lists of pages read (*read notices*) to message types that already carry analogous information about pages written, and (iii) we added an extra message round at barriers in order to retrieve word-level access information, if necessary.

We use the ATOM [21] code-rewriter to instrument shared accesses with calls to analysis routines. ATOM allows executable binaries to be analyzed and modified. We use ATOM to identify and instrument all loads and stores that may access shared memory. Although ATOM is available only for DEC Alpha systems, similar tools are becoming more common. EEL [13] provides similar support for Sparc and MIPS systems, and several machine vendors are working on such tools as well.

The actual instrumentation consists of a procedure call

to an analysis routine that sets a bit in a per-page bitmap if the instruction accesses shared memory. Information about which pages were accessed, together with the bitmaps themselves, is placed in known locations for CVM to use during the execution of the application. All data structures, including bitmaps, are statically allocated in order to reduce runtime cost.

The overall procedure for detecting data races is the following:

1. CVM synchronization messages carry information about process intervals. Each interval contains one or more *write notices* that specify pages written during that interval. We augmented interval structures to also carry *read notices*, or lists of pages read during that interval. Interval structures also contain version vectors that identify the logical time associated with the interval, and permit checks for concurrency.
2. Worker processes in any LRC system append consistency information describing all local intervals to barrier arrival messages. At each barrier, therefore, the barrier master has complete and current information on all intervals in the entire system. This information is sufficient for the master to locally determine the set of all pairs of concurrent intervals. Although the algorithm must potentially compare the version vector of each interval of a given processor with the vector of each interval of every other processor, synchronization and program order allow many of the comparisons to be bypassed. Version vector comparison is a constant time process, requiring only two integer comparisons.
3. For each pair of concurrent intervals, the read and write notices are checked for overlap. A data race might exist on any page that is either written in two concurrent intervals, or read in one interval and written in the other. Such interval pairs, together with a list of overlapping pages, are placed on the *check list*.
4. Barrier release messages carry the check list to all system processes. Each read or write notice has a corresponding bitmap that describes precisely which words of the page were accessed. These bitmaps are returned

to the barrier master for each page and interval on the check list.

5. The barrier master compares bitmaps from overlapping pages in concurrent intervals. Bitmap comparison is a constant time process, dependent on page size. In the case of a read-write or write-write overlap, the algorithm has determined that a data race exists, and prints the address of the affected variable.

We currently use a very simple interval comparison algorithm to find pairs of concurrent intervals, primarily because the major system overhead is elsewhere. The upper bound on the number of intervals per processor pair that the comparison algorithm must compare is $O(i^2)$, where i is the maximum number of intervals of a single processor since the last barrier. The algorithm needs only to examine intervals created during the last barrier epoch. By definition, these intervals are separated from intervals in previous epochs by synchronization, and are therefore ordered with respect to them. Since each interval potentially needs to be compared against every other interval (of another process in the current epoch), the total comparison time per barrier is bounded by $O(i^2 p^2)$, where p is the number of processes. In practice, however, the number of comparisons is usually quite small.

Applications that use only barriers have two intervals per process per barrier epoch. More than two intervals per barrier are only created through additional peer-to-peer synchronization, such as exclusive locks. However, peer-to-peer synchronization also imposes ordering on intervals of the synchronizing processes. For example, a lock release and subsequent acquire order intervals prior to the release with respect to those subsequent to the acquire. Since an ordered pair of intervals can not be concurrent, the same act that creates intervals also removes many interval pairs from consideration for data races. Hence, programs with many intervals between barriers usually also have ordering constraints that reduce the number of concurrent intervals.

5 Performance

We evaluated the performance of our prototype by searching for data races in implementations of four common shared-memory applications: FFT (Fast Fourier Transform), SOR (Jacobi relaxation), TSP (branch and bound traveling salesman problem), and Water (a molecular dynamics simulation from the Splash2 [22] benchmark suite. All applications were run on DECstations with four 250 Mhz Alpha processors, connected by a 155 MBit ATM. We used only a single processor per machine in order to avoid bus contention.

Table 1 summarizes the application inputs, synchronization types, the number of intervals per barrier, and the overall

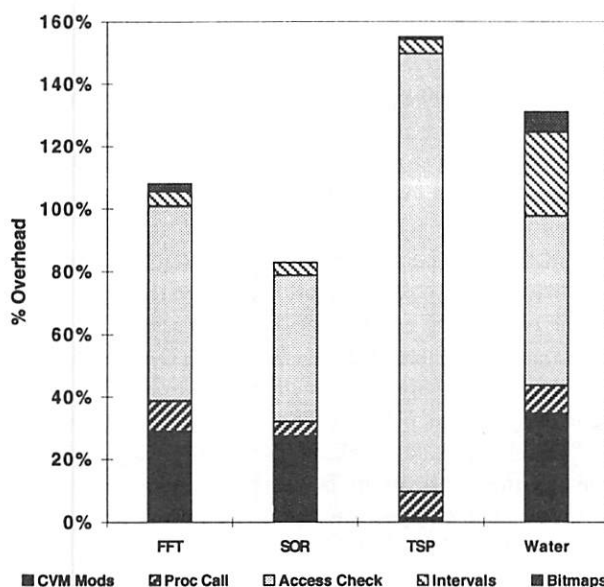


Figure 3. Overhead Breakdown

slowdown for eight-processor runs. "Memory size" is the size of the shared data segment. "Intervals Per Barrier" is the average number of intervals created between barriers. As the number of interval comparisons is potentially proportional to the number of intervals squared, this metric gives a rough idea of the worst-case cost of running the comparison algorithm. This number is greater than 1 for FFT and SOR because our barrier implementation requires two interval structures per barrier. Other synchronization mechanisms require only a single interval per synchronization. As the next section will show, the comparison algorithm is at most only the third most costly form of overhead in our applications.

"Slowdown" is the runtime slowdown for each of the applications, compared with an uninstrumented version of the application running on an unaltered version of CVM. Over the four applications, execution time slows only by an average factor of 2.2. This number compares quite favorably even with systems that exploit extensive compiler analysis.

Figure 3 shows the overhead added by the race-detection mechanism relative to the running time of the uninstrumented binary, for each application. For example, the execution time of the instrumented FFT binary is 108% longer than that of the uninstrumented binary. "CVM Mods" is the overhead added by the modifications to CVM, primarily setting up the data structures necessary for proper data-race detection and the additional bandwidth used by the read notices. "Proc Call" is the procedure call overhead for our instrumentation. ATOM will not currently inline instrumentation; only procedure calls can be inserted into existing code. The ATOM team is working to eliminate this restriction, and the "Proc

Call" column shows how much of the total overhead could be eliminated as a result. "Access Check" is the additional time spent inside the procedure call determining whether an access is to shared memory, and setting the proper bit if so. "Intervals" refers to the time spent using the interval comparison algorithm to identify concurrent interval pairs with overlapping page accesses. "Bitmaps" describes the overhead of the extra barrier round required to retrieve bitmaps, together with the cost of the bitmap comparisons.

The two largest components of the overhead are the access checks and modifications to CVM. The overheads of the interval comparison algorithm and the bitmap checks are usually fairly small. As we will see in the next section, TSP has a higher rate of calls to the runtime analysis routines than the other applications, hence the higher instrumentation overhead. The comparison algorithm adds more overhead for Water than the other applications because of the large degree of fine-grained synchronization.

The following subsections describe the above overheads in more detail.

5.1 Instrumentation Costs

We instrumented each load and store that could potentially be involved in a data race. The instrumentation consists of a procedure call to an analysis routine, and hence adds "Proc Call" and "Access Check" overheads. By summing these columns from Figure 3, we can see that instrumentation accounts for an average of 68% of the total race-detection overhead.

This overhead can be reduced by instrumenting fewer instructions. This goal is difficult because shared and private data are all accessed using the same addressing modes, and even share some base registers. However, we eliminate most stack accesses by checking for use of the frame pointer as a base register. The fact that all shared data in our system is dynamically allocated allows us to eliminate any instructions that access private data by indirection through the base register that points to statically allocated data. Finally, we do not instrument any instructions in shared libraries because none of our applications pass segment pointers to any libraries. This is the case with the majority of the scientific programs where data race detection is most important. However, we can easily instrument "dirty" library functions, if necessary.

Table 2 breaks down load and store instructions into the categories that we are able to statically distinguish. The first four columns show the number of loads and stores that are not instrumented because they access the stack or statically-allocated data, or are in library routines, including CVM itself. The fifth column shows the remainder. These instructions could not be eliminated as possible data-race participants and are therefore instrumented by ATOM to

App	Load and Store Instructions				
	Stack	Static	Library	CVM	Inst.
FFT	1285	1496	124716	3910	261
SOR	342	1304	48717	3910	126
TSP	244	1213	48717	3910	350
Water	649	1919	124716	3910	528

Table 2. Instrumentation Statistics

make a procedure call to an analysis routine each time the memory access is executed.

On average, we are able to statically determine that over 99% of the loads and stores in our applications are to non-shared data. As an example, the FFT binary contained 131,668 load and store instructions. Of these, 124,716 instructions are in libraries. A further 1285 instructions access data through the frame pointer, and hence reference stack data. Another 3910 are in the CVM system itself. Finally, 1496 instructions access data through a register pointing to the base of statically allocated global memory. We can eliminate these instructions as well, since CVM allocates all shared memory dynamically. In the entire binary, only 261 memory access instructions remain that could possibly reference shared memory, and hence form part of a data race.

Nonetheless, the last two columns of Table 3 show that the majority of run-time calls to our analysis routines are for private, not shared, data. "Inst. Accesses Per Second" refers to the number of instrumented loads and stores executed per second, and the number of these calls to our instrumentation routines that turn out to be for shared or private data. The high rate of instrumented accesses for TSP explains the large "Access Check" overhead for TSP in Figure 3. Accesses to shared data are distinguished from accesses to private data by comparing the address with that of the shared data segments.

5.2 The Cost of the Comparison Algorithm

The comparison algorithm has three tasks. First, the set of concurrent interval pairs must be found. Second, this list must be winnowed down to those interval pairs for which an overlap of pages is found (i.e. one interval of a pair reads from page x and the other interval in the pair writes to page x). Each such pair of concurrent intervals exhibits unsynchronized sharing. However, the sharing may be either false sharing, i.e. the loads and stores to page x are to different locations in x (not a data race), or true sharing, in which the loads and stores overlap at least one location (data race).

The first column of Table 3 shows the percentage of intervals that are involved in at least one such concurrent interval pair. This number ranges from zero for SOR, where

	Intervals Used	Bitmaps Used	Msg Overhead	Inst. Accesses Per Second	
				Shared	Private
FFT	15%	1%	0.4%	311079	924226
SOR	0%	0%	1.6%	483310	251200
TSP	93%	13%	1.3%	737159	2195510
Water	13%	11%	48.3%	145095	982965

Table 3. Dynamic Metrics

this is no unsynchronized sharing (true or false), to 93% for TSP, where there is a large amount of both true and false sharing. Note that the number of possible interval pairs is quadratic with respect to the number of intervals, so even if this stage eliminates only 7% of all intervals, as we do for TSP, we may be eliminating a much higher percentage of interval pairs.

The second column of Table 3 shows that an average of only 6% of all bitmaps must be retrieved from constituent processors in order to identify data races by distinguishing false from true sharing. As page access lists of concurrent intervals will only overlap in cases of false sharing or actual data races, the percentage of intervals and bitmaps involved in comparisons is fairly small.

5.3 The Cost of CVM Modifications

Figure 3 shows that almost 22% of our overhead comes from “CVM Mods”, or modifications made to the CVM system in order to support the race-detection algorithm. This overhead consists of the cost of setting up additional data structures for data-race detection, and the cost of the additional bandwidth consumed by read notices.

The third column of Table 3 shows the bandwidth overhead of adding read notices to synchronization messages. Individual read and write notices are the same size, but there are typically at least five times as many reads as writes, and read notices consume a proportionally larger amount of space than write notices. The bandwidth overhead for Water is much larger than for the other applications because of the fine-grained synchronization, and hence the large number of intervals.

The bandwidth consumed by read notices prevents us from running larger input sets because current message sizes are already at system maximums. Our communication code can, and eventually will, be modified in order to accommodate larger messages.

6 Discussion

6.1 Reference Identification

The system currently prints the shared segment address for each detected race condition, together with the interval indexes. In combination with symbol tables, this information can be used to identify the exact variable and synchronization context.

Identifying the specific instructions involved in a race is more difficult because it requires retaining program counter information for each shared memory access. This information is available at runtime, but such a scheme would require saving program counters for each shared access until a future barrier analysis phase determined that the access was not involved in a race. The storage requirements for retaining this information would generally be prohibitive, and would also add runtime overhead.

A second approach is to use the conflicting address and corresponding barrier epoch from an initial run of the program as input to a second run. During the second run, program counter information can be gathered for only those accesses to the conflicted address that originate in the barrier epoch determined to involve the data race.

While runtime overhead and storage requirements can thereby be drastically reduced, the data race must occur in the second run exactly as in the first. This will happen if the application has no *general races* [18], i.e. synchronization order is deterministic. This is not the case in either of the two applications for which we found data races. A solution is to modify CVM so as to save synchronization ordering information from the first run, and to enforce the same ordering in the second run.

6.2 Scalability

Figure 4 shows runtime slowdown versus number of processors. Slowdown actually decreases as we increase the number of processors. This seemingly anomalous result has two causes. First, interval and bitmap comparison overhead is serialized at the master process, and hence *observable* overhead from these sources remains constant as system

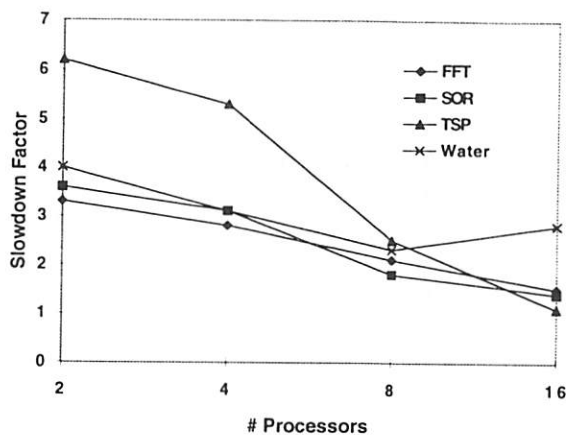


Figure 4. Slowdown Factor versus Number of Processors

size is increased. Instrumentation costs, however, occur in parallel with the shared accesses. As system size increases, therefore, per-process computation and observable instrumentation overhead decreases.

Second, the combination of modest problem sizes, fast processors, and the large page size of the DECstations result in our applications getting very modest speedups even with the unmodified version of the single-writer protocol used in this study. Hence, at least some of the overhead of the race detection algorithm is probably masked by DSM overhead. However, none of these limitations are intrinsic to our approach. Our problem sizes are small because of message size limitations. We are modifying the underlying communication layer to alleviate this problem. The large page size exacerbates the problems of false sharing associated with single-writer protocols [12] protocols. We based our prototype on CVM's single-writer protocol in order to minimize complexity, but our algorithm will work identically with CVM's multi-writer protocol.

Finally, comparison to determine if two intervals are concurrent is a constant-time process, as each interval is marked with a vector timestamp [14, 10]. Comparison of two concurrent intervals to determine whether their page lists overlap is currently $O(n^2)$ in the size of the lists, as they are usually very small (i.e. less than ten). If we encountered applications where these lists grew large, we could perform the comparison in time linear with respect to the number of pages in the system by implementing page lists using bitmaps.

6.3 Global Synchronization

The interval comparison algorithm is currently run only at global synchronization operations, i.e. barriers. The ap-

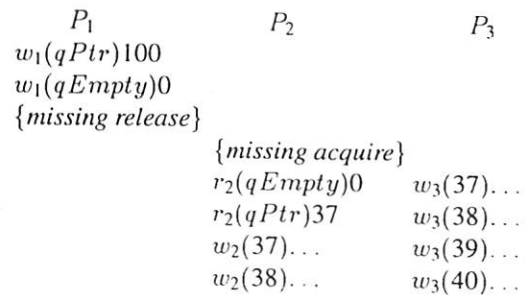


Figure 5. The race $w_2(37) - w_3(37)$ would not occur in SC system

plications and input sets in this study use barriers frequently enough, or otherwise synchronize infrequently enough, that the number of intervals to be compared at barriers is quite manageable. Nonetheless, there certainly exist applications for which global synchronization is not frequent enough to keep the number of interval comparisons to a small number. Ideally, the system would be able to incrementally discard data races without global cooperation, but such mechanisms would increase the complexity of the underlying consistency protocol [8]. If global synchronization is either not used, or not used often enough, we can exploit CVM routines that allow global state to be consolidated between synchronizations. Currently, this mechanism is only used in CVM for garbage collection of consistency information in long-running, barrier-free programs.

6.4 Accuracy

Adve [2] discusses three potential problems in the accuracy of race detection schemes in concert with *weak memory* systems, or systems that support memory models such as lazy release consistency.

The first issue is whether to report all data races, or only those that would also occur during sequentially-consistent executions of the program. Their example (somewhat simplified) is shown in Figure 5, where the notation $op(loc) val$ indicates a read or write operation performed on location loc , that respectively reads or writes value val . If the missing synchronization operations were present, there would not be any races. P_2 's read of $qPtr$, $r_2(qPtr)37$, would return 100 instead of some older value (37 in this case), and P_2 's subsequent writes would be to locations 100 and above. However, given that the synchronization is not present, only the $qPtr$ and $qEmpty$ races would have occurred on sequentially consistent hardware. If $w_1(qEmpty)0$ had propagated to P_2 , any sequen-

tially consistent system must also have sent the results of $w_1(qPtr)100$. This is not the case with weak memory systems, which can usually reorder the effects of write operations between synchronization points at will. Hence, the races between $w_2(37)$ and $w_3(37)$, etc., only occur on weak memory systems.

This is an instance of a more general problem, i.e. whether to return all data races, or only “first” data races, those that are not affected or caused by any prior race. Our system currently reports all races, but could be modified to report only first races without requiring more information to be gathered. Determining whether one race is affected by another effectively consists of deciding whether a happens-before-1 relationship exists between any of the operations in one race and any of the operations in another. Since barrier operations are semantically equivalent to releases by all arriving processors to the barrier master, followed by the barrier master releasing to all other processors, any race in a prior barrier epoch must necessarily affect all races in subsequent epochs. Hence, all “first” races must occur in the same barrier epoch. Modifying our system to perform this check online is a trivial extension.

The second problem with accuracy of dynamic race-detection algorithms is reliability of ordering information in the presence of races. Race conditions could cause wild accesses to random memory locations, potentially corrupting interval ordering information or access bitmaps. This problem exists in any dynamic race-detection algorithm, but we expect it to occur infrequently.

A final accuracy problem identified by Adve is that of systems that attempt to minimize space overhead by buffering only limited trace information, possibly resulting in some races remaining undetected. Our system only discards trace information when it has been checked for races, and hence does not suffer this limitation.

6.5 Further Performance Enhancements

There are several ways that overhead can be further reduced. First, the ATOM team has promised a new version that allows instrumentation code to be inlined. The Shasta project [20] has already demonstrated a version of ATOM with this feature. Figure 3 shows that an average of 6.7% of our overhead is caused by the procedure call. This overhead will be eliminated when we get the new version of ATOM.

Second, we currently instrument both load and store instructions. This is necessary because our system is currently built on top of a single-writer LRC protocol [12]. Converting our system to use the multi-writer protocol would allow us to exploit existing *diffs*, which summarize per-page modifications, to extract write accesses. We would then be able to dispense with the monitoring of store instructions. Since approximately 68% of the overhead is from instrumentation,

and 25% of all data accesses are stores, we should be able to eliminate at least 17% of overall overhead.

A disadvantage of this approach is a slightly weaker correctness guarantee. *Diff*s only contain *modifications* to shared data. If a shared value is overwritten with the same value, the data location will not be in the *diff*, and any data race involving this location may not be detected.

Finally, Table 2 shows that nearly 68% of the calls to our instrumentation routines turn out to be for private data. Our current analysis tracks references only through the same basic block. If the value defined before that point is used to reference an unknown data location, we conservatively assume that the location is shared, and hence instrument the access. Inter-procedural analysis would allow us to eliminate many of these “false” instrumentations, and reduce overall overhead. This analysis can be done with the current ATOM system, but will be much easier with a version promised in the near future.

7 Related Work

There has been a great deal of published work in the area of data race detection. However, as previously mentioned, most prior work has dealt with applications and systems in more specialized domains. Bitmaps have been used to track shared accesses before [5], but we know of no other implementation of on-the-fly data-race detection for explicitly-parallel, shared-memory programs without compiler support.

Our work is closely related to work already alluded to in Section 5, a technique described (but not implemented) by Adve et al. [2]. The authors describe a post-mortem technique that creates trace logs containing synchronization events, information allowing their relative execution order to be derived, and computation events. Computation events correspond roughly to CVM’s intervals. Computation events also have READ and WRITE attributes that are analogous to the read and write page lists and bitmaps that describe the shared accesses of an interval. These trace files are used off-line to perform essentially the same operations as in our system. We differ in that our minimally-modified system leverages off of the LRC memory model in order to abstract this synchronization ordering information *online*. We are therefore able to perform all of the analysis online as well, and do away with trace logs, post-mortem analysis, and much of the overhead.

We have also just become aware of unpublished work on execution replay in TreadMarks that could be used to implement race-detection schemes. The approach of the Reconstruction of Lamport Timestamps (ROLT) [19] technique is similar to the technique we described in Section 6.1 for identifying the instructions involved in races. Minimal ordering information saved during an initial run is used to

enforce exactly the same interleaving of shared accesses and synchronization in a second run. During the second run, a complete address trace can be saved for post-mortem analysis, although the authors do not discuss race detection in detail. The advantage of this approach is that the initial run incurs minimal overhead, ensuring that the tracing mechanism does not perturb the normal interleaving of shared accesses.

The ROLT approach is complementary to the techniques described in this paper. The primary thrust of our work is in using the underlying consistency mechanism to prune enough information *online* that post-mortem analysis is not necessary. As such, our techniques could be used to improve the performance of the second phase of the ROLT approach. Similarly, our system could be augmented to include an initial synchronization-tracing phase, allowing us to reduce our perturbation of the parallel computation.

8 Conclusions

We have presented a new on-the-fly data race detection technique that allows data-race detection in explicitly-parallel, shared-memory programs. Our technique abstracts synchronization ordering from consistency information already maintained by lazy-release-consistent DSM systems. We are able to use this information to eliminate most access comparisons, and to perform the entire data-race detection online.

The primary costs of data-race detection in our system are in tracking shared data accesses. We use ATOM to instrument load and store instructions with calls to our library. We are able to statically eliminate more than 99% of all loads and stores in binaries by identifying accesses to stack variables and statically-allocated global variables. Nonetheless, the majority of the runtime calls to our library are for non-shared accesses. Overall, our applications slow down by an average factor of approximately two.

We used our system to analyze four shared-memory programs, finding data races in two. One of the programs, TSP, allows data races in order to improve performance without violating correctness. The data race in the other program, which was from a standard benchmark suite, was a bug. We believe that the utility of our techniques, in combination with the generality of the programming model that we support, can help data-race detection to become more widely used.

Additional information on CVM is available at: <http://www.cs.umd.edu/projects/cvm.html>.

9 Acknowledgments

We would like to thank Carla Ellis and the anonymous referees for all their helpful suggestions and feedback on

earlier drafts of this paper.

References

- [1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.
- [2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243, May 1991.
- [3] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *International Conference on Parallel Processing*, pages 721–727, August 1987.
- [4] J. Choi and S. L. Min. Race frontier: Reproducing data races in parallel program debugging. In *Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming*, April 1991.
- [5] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 1–10, March 1990.
- [6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
- [7] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings Supercomputing '90*, pages 15–26, May 1990.
- [8] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, 1994.
- [9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
- [10] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

- [11] Pete Keleher. The Coherent Virtual Machine. Technical Report Maryland TR93-215, Department of Computer Science, University of Maryland, September 1995.
- [12] Pete Keleher. The relative importance of concurrent writers and weak consistency models. To appear in *The Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.
- [13] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.
- [14] F. Mattern. Virtual time and global states of distributed systems. In *Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers, Amsterdam, 1989.
- [15] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. Technical Report CRPC-TR92232, Rice University, September 1992.
- [16] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming*, April 1991.
- [17] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *1990 International Conference on Parallel Processing*, pages 93–97, August 1990.
- [18] Robert H. B. Netzer and Barton P. Miller. What are race conditions? In *ACM Letters on Programming Languages and Systems*. ACM, March 1992.
- [19] M. A. Ronsse and W. Zwaenepoel. Execution replay for TreadMarks. Submitted for publication, 1996.
- [20] Daniel Scales and Kourosh Gharachorloo. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, May 1994.
- [22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

Lightweight Logging for Lazy Release Consistent Distributed Shared Memory

Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, Miguel Castro

IST - INESC

R. Alves Redol 9, 1000 Lisboa PORTUGAL

{msc, pjg, mds, nuno, miguel}@inesc.pt

Abstract

This paper presents a new logging and recovery algorithm for lazy release consistent distributed shared memory (DSM). The new algorithm tolerates single node failures by maintaining a distributed log of data dependencies in the volatile memory of processes.

The algorithm adds very little overhead to the memory consistency protocol: it sends no additional messages during failure-free periods; it adds only a minimal amount of data to one of the DSM protocol messages; it introduces no forced rollbacks of non-faulty processes; and it performs no communication-induced accesses to stable storage. Furthermore, the algorithm logs only a very small amount of data, because it uses the log of memory accesses already maintained by the memory consistency protocol.

The algorithm was implemented in TreadMarks, a state-of-the-art DSM system. Experimental results show that the algorithm has near zero time overhead and very low space overhead during failure-free execution, thus refuting the common belief that logging overhead is necessarily high in recoverable DSM systems.

1. Introduction

Networks of workstations are a cost-effective alternative to dedicated parallel computers for running compute-intensive applications [Anderson 95]. In these networks, the simplicity of programming with a shared memory abstraction can be retained by using distributed shared memory systems [Li 86, Keleher 94]. As these networks grow and the number of nodes running a DSM application increases, the probability of a failure also increases. This probability can become unacceptably high, especially for long-running applications. Therefore, DSM systems need mechanisms to tolerate faults in computing nodes.

This paper presents a new logging and recovery algorithm for a DSM system based on lazy release

consistency (LRC) [Keleher 92]. This algorithm is inspired in our previous work where an approach with comparable properties was applied to an entry-consistent memory system [Neves 94]. We extended TreadMarks [Keleher 94], an efficient implementation of LRC, to implement the logging and recovery algorithm. Experimental results show that the algorithm has near zero time overhead and very low space overhead during failure-free execution, thus refuting the common belief that logging overhead is necessarily high in recoverable DSM systems [Cabillac 95, Suri 95]. We show that the logging overhead of the algorithm is much lower than the previously proposed logging algorithms for recovery of DSM systems based on LRC [Suri 95].

The algorithm tolerates single node failures by maintaining a distributed log of data dependencies in the volatile memory of processes. Whenever a process sends relevant information to another, it keeps in its volatile memory a log of that information. The algorithm achieves very good performance because it is tightly integrated with the LRC memory coherence protocol. Since the LRC protocol already keeps most of the relevant information in its data structures, the algorithm only has to log the instants at which the information is transferred. Whenever a remote synchronization operation occurs, it logs a pair with the vector times of the sender and receiver in the volatile memories of both processes. The algorithm does not log the value of shared memory updates, because it takes advantage of the log of memory accesses already maintained by the memory consistency protocol. This results in significantly reduced space overhead when compared to previously proposed solutions [Suri 95]. Furthermore, the algorithm sends no additional messages and sends only a very small amount of extra data, during failure-free periods, because most of the information necessary for recovery is already part of the LRC protocol messages. Only one of the LRC protocol messages is modified to include an additional integer.

Periodically, each process takes a checkpoint, independently from the other processes, where it records in stable storage all its state. Unlike other proposals [Janssens 93, Suri 95], our logging algorithm performs no communication-induced accesses to stable

Manuel Costa, Nuno Neves and Miguel Castro were supported in part by Praxis XXI fellowships from JNICT.

storage; independent checkpoints are used only to speed up recovery. When a process crashes, the system detects that it crashed and restarts the process on an available processor, using its latest saved checkpoint. The recovering process re-executes with information gathered from the logs of the surviving processes; the non-faulty processes are not forced to rollback after a failure. We assume processes execute in a piecewise deterministic manner [Strom 85] and the only non-deterministic events are shared memory reads. Therefore, the system will recover to a consistent state if the reads of the faulty process can be replayed to return the same values they returned before the crash. This is achieved by providing the recovering process with a sequence of page updates from other processes that is equivalent to the one received before the crash.

The logging algorithm is combined with a consistent checkpointing scheme, which allows multiple failures to be tolerated. It supports the more probable single failures with low overhead, and takes consistent checkpoints infrequently to tolerate the less probable multiple failures. Therefore, the combined algorithm is a kind of two-level recovery scheme as described in [Vaidya 95], where it was shown that this type of algorithm can achieve better performance than one-level recovery schemes. The consistent checkpointing scheme is integrated with the global garbage collection operations already performed by the LRC protocol, hence introducing no additional coordination overhead. During each garbage collection operation, all data structures needed for recovery are discarded and a consistent checkpoint is created. Although consistent checkpointing could also be used to recover from single failures, using the logging and recovery algorithm provides several benefits. Single process faults are recovered efficiently because only the failed process needs to rollback. Some applications can still make progress while one of the processes is recovering, and during recovery only the recovering process consumes resources. Recovery is potentially faster since idle time due to remote synchronization operations is eliminated.

Our implementation required only small changes to the existing TreadMarks implementation of LRC. Code was added to log dependencies and handle recovery. We retained the most significant optimizations used by TreadMarks, such as lazy diff and interval creation [Keleher 94]. In summary, this paper makes the following contributions:

- It describes a new logging and recovery algorithm for lazy release consistent DSM
- It explains how to integrate logging with the memory consistency protocol without sending additional messages, sending only a small amount of extra data, and logging a minimal amount of data

- It presents an implementation of the algorithm using a state-of-the-art LRC-based DSM system
- It presents results showing that the algorithm is very space and time efficient, refuting the common belief that logging overhead is necessarily high for recoverable DSM systems.

The remainder of the paper is organized as follows. The next section presents related work. Section 3 describes TreadMarks and lazy release consistency. Section 4 explains the logging and recovery algorithm. Section 5 presents experimental results showing the overhead introduced by our logging mechanism, and in section 6 we draw our conclusions.

2. Related Work

Several systems have used consistent or independent checkpointing to transparently recover from transient faults. In consistent checkpointing [Chandy 85, Koo 87] processes execute a coordination protocol to establish a new consistent global state. We use consistent checkpointing only to recover from multiple concurrent failures. Cabillic et al. [Cabillic 95] integrate consistent checkpointing with barrier crossings, simplifying the coordination protocol performed by the nodes of the system. We integrate consistent checkpointing with the LRC garbage collection protocol, introducing no additional coordination overhead.

In independent checkpointing [Johnson 87, Strom 85] there is no coordination, but communication-induced checkpoints are performed or some logging has to be done during failure-free periods to prevent rollback propagation during recovery. In sender-based message logging [Johnson 87, Johnson 89], whenever a process sends a message, it logs it in its volatile memory; when the message is received, the receiver returns a sequence number indicating the order of receipt, which the sender adds to its volatile log. The state of a failed process can then be recovered by replaying the logged messages from the distributed log at the surviving processes. We also log recovery information in the volatile memory of sender processes, and use that information to replay messages during recovery. However, message logging protocols are not directly applicable to DSM systems because they do not handle asynchronous message reception [Suri 95] and it has been shown that some potential message passing dependencies can be eliminated in DSM systems [Janssens 94, Janakiraman 94]. Our algorithm takes advantage of this kind of optimization since it is tightly integrated with the DSM consistency protocol and logs only the minimal information required for recovery.

Wu and Fuchs [Wu 90] proposed the first recovery algorithm for sequentially consistent DSM. In their

checkpointing scheme, a process is required to save its state whenever a modified page is sent to another process. Janssens and Fuchs [Janssens 93] introduced checkpointing for relaxed memory models, proposing that a node should checkpoint its state whenever it releases or acquires a lock.

Richard and Singhal [Richard III 93] used logging to recover a sequentially consistent DSM from single node failures. In their protocol, shared pages are logged in volatile memory whenever they are read. The log is flushed to stable storage before transferring a modified page to another process. Suri, Janssens and Fuchs [Suri 95] improved this proposal by noting that accesses to shared memory need not be logged but only tracked. For lazy release consistency, they proposed logging the messages received at acquire points and access misses. The log is flushed to stable storage whenever page invalidates or page updates are sent to another process.

Our recovery algorithm for lazy release consistency improves this by logging less data and not requiring communication-induced flushing of the log to stable storage. On the other hand, in our algorithm it is necessary to involve operational processes in the recovery of a failed process. However, operational processes can continue their execution while the failed process is recovering. Our recovery algorithm keeps some of the properties of the protocol previously proposed by Neves, Castro and Guedes [Neves 94]. This protocol was designed to recover DiSOM [Castro 96], a multi-threaded object-based entry consistent DSM system based on an update protocol.

While the protocols described above recover the state of the shared memory as well as the execution state of the processes, some protocols only recover the shared memory state. Stumm and Zhou [Stumm 90] proposed a protocol which tolerates single node failures by ensuring that each page is kept in the memory of at least two nodes of the DSM system. Feeley et al. [Feeley 94] developed a transactional distributed shared memory where coherency is integrated with a mechanism for recoverability of persistent data. The protocol proposed by Kermarrec et al. [Kermarrec 95] establishes a recovery point by creating two recovery copies of every modified page and storing them at distinct nodes. A two-phase commit protocol is used to atomically update a recovery point.

3. LRC and TreadMarks

TreadMarks [Keleher 94] implements a relaxed memory model called lazy release consistency [Keleher 92]. TreadMarks ensures that all programs without data races behave as if they were executing on a conventional sequentially consistent (SC) memory. Most programs satisfy this condition and behave

identically in both models, but LRC has the advantage that it can be implemented more efficiently. This section describes TreadMarks' implementation of LRC without the extensions we added to provide fault tolerance.

LRC divides the execution of each process into logical intervals that begin at each synchronization access. Synchronization accesses are classified as *release* or *acquire* accesses. Acquiring a lock is an example of an acquire, and releasing a lock is an example of a release. Waiting on a barrier is modeled as a release followed by an acquire. LRC defines the relation *corresponds* on synchronization accesses as follows: a release access on a lock corresponds to the next acquire on the lock to complete (in real time order); and a release access on a barrier wait corresponds to the acquire accesses executed by all the processes on the same barrier wait.

Intervals are partially ordered according to the transitive closure of the union of the following two relations: (i) intervals on a single process are totally ordered by program order; and (ii) an interval *x* precedes an interval *y*, if the release that ends *x* *corresponds* to the acquire that starts *y*. The partial order between intervals is represented by assigning a vector timestamp to each interval. TreadMarks implements lazy release consistency by ensuring that if interval *x* precedes interval *y* (according to this partial order), all shared memory updates performed during *x* are visible at the beginning of *y*.

3.1 Data Structures

Each process in TreadMarks maintains the following data structures in its local volatile memory:

pageArray: array with one entry per shared page.

procArray: array with one list of interval records per process.

dirtyList: identifiers of pages that were modified during the current interval.

VC: local vector clock.

pid: local process identifier.

Each **SharedPageEntry** has fields:

status: operating system protection status for page (no-access, read-only or read-write)

twin: original copy of the page

writeNotices: array with one list of write notice records per process

manager: identification of the page manager

copyset: set of processes with copy of the page

Each **WriteNoticeRecord** has fields:

diff: pointer to diff

interval: pointer to corresponding interval record

pageID: page number

Each **IntervalRecord** has fields:

- idCreat:** id of process which created the interval
- vc:** vector time of creator
- writeNotices:** list of write notice records for this interval.

The status field of a page entry is the operating system protection status for the page, i.e. if the status is no-access then any access to the page triggers a page fault, and if the status is read-only a write access to the page triggers a page fault. The writeNotices field in the page entry describes modifications to the page. The entry for process i in the writeNotices array contains a list with all the write notices created by i for the page, that are known to the local process. Each of these write notice records describes the updates performed by i to the page in a given interval. The write notice record contains a pointer to the interval record describing that interval, and a pointer to a diff containing the words of the page that were updated in the interval. The interval records contain a backpointer to a list with one write notice for each page that was modified during the interval. Whenever an interval record is created, it is tagged with the vector time and the identity of its creator.

The *procArray* has an entry for each process. The entry for process i contains a list of interval records describing the intervals created by i that the local process knows about. This list is ordered by decreasing interval logical times. We refer to the value of $VC_i[i]$ as i 's vector time, and to the value of $VC_i[i]$ as i 's logical time. Similarly, the vector time of an interval created by i is the value of VC_i when the interval is created, and the logical time of the interval is the value of $VC_i[i]$.

3.2 Memory Consistency Algorithm

This subsection describes the implementation of TreadMarks' memory consistency algorithm. The description is based on the pseudo-code for this algorithm presented in Figure 3.1.

LockAcquire is executed when a process tries to acquire a lock and it was not the last process to hold the lock. It sends a request message to the lock manager. The message contains the current vector time of the acquirer. The lock manager forwards the message to the last acquirer.

LockAcquireServer is executed by the releaser when it receives the lock request message. If a new interval has not yet been created since the last local release of this lock, the releaser creates a new interval record for the current interval; and for all the pages modified during the interval it creates write notice records. The diffs encoding the modifications to each of these pages are not created immediately. Instead, they are created

lazily when the process receives a diff request or a write notice for a page. The reply message includes a description of all the intervals with timestamps between the acquirer's and the releaser's vector times. We say that an interval i , created by process p , is between the acquirer's and the releaser's vector times if $VC_{acq}[p] < i.vc[p] \leq VC_{rel}[p]$.

The description of each interval contains the identifier of the process that created the interval, the vector timestamp of the interval, and the corresponding write notices (remember that interval record = [idCreat, vc, writeNotices]). Each write notice in the reply contains only the number of a page that was modified during the interval; no updates are transferred in this message. The information in the reply is obtained by traversing the lists of interval records in *procArray*. In the reply, each sequence of intervals created by process p is ordered by increasing logical time of p .

The acquirer calls IncorporateIntervals to incorporate the information received in the reply in its own data structures. Notice that the system creates a diff for writable pages for which a write notice was received. This is important to allow the system to distinguish the modifications made by concurrent writers to the page [Carter 91]. The status of the pages for which write notices were received in the reply is set to no-access. When the acquirer tries to access one of the invalidated pages the system invokes the PageFaultHandler. On the first fault for a page, a page copy is requested from the page manager. Pages are kept writable at the manager until the first page request arrives. When this happens, the page status is changed to read-only. After getting the page copy, if there are write notices for the page without the corresponding diffs, the system sends messages requesting those diffs, to the processes that cache them. In TreadMarks, a processor that modified a page in interval i is guaranteed to have all the diffs for that page for all intervals preceding i . After receiving the diffs, the handler applies them to the page in timestamp order. On a read miss, the system provides read-only access to the page. On a write miss, the system provides read-write access and creates a copy of the page (a "twin"), which is used to detect modifications to the page. The twin is later compared to the current contents of the page to produce a diff that encodes the modifications produced during the interval.

The Barrier routine is executed by the system when a process waits on a barrier. If a process is not the manager of the barrier, it sends to the manager its current vector time, plus all intervals between the logical time of the last local interval known at the manager and its current logical time. After this, the manager sends to each other process all the intervals between the current vector time of the process and the

LockAcquire:

```
send (AcqRequest, VC) to lock manager;
receive (intervals)
IncorporateIntervals(intervals);
```

LockAcquireServer:

```
receive (AcqRequest, VCacq) from acquirer;
wait until lock is released;
if (an interval was not created since the
    last release of this lock)
    CreateInterval;
send (intervals between VCacq and VC) to
    acquirer;
```

LockRelease:

```
if (lock request pending)
    wakeup LockAcquireServer;
```

CreateInterval:

```
if (dirtyList is not empty) {
    VC[pid] := VC[pid] + 1;
    insert new interval i in
        procArray[pid];
    for each page in dirtyList
        create write notice record for
            interval i;
    clear dirtyList;
}
```

IncorporateIntervals(intervals):

```
for each i in intervals {
    insert record for i in
        procArray[i.idCreat];
    VC[i.idCreat] := i.vc[i.idCreat];
    for each write notice in i {
        store write notice;
        if (twin exists for the page write
            notice refers to) {
            if (a write notice corresponding
                to the current writes was not
                already created)
                CreateInterval;
            create diff;
            delete twin;
        }
        set page status to no-access;
    }
}
```

DiffRequestServer:

```
receive (DiffRequest, pageId, diffId)
    from req node;
if (diff is not created) {
    create diff;
    delete twin;
    set page status to read-only;
}
send (diff) to req node;
```

PageFaultHandler:

```
if (page status = no-access) {
    if (local page copy not initialized) {
        send (PageRequest, pageId)
            to page manager;
        receive (page copy, copyset);
    }
    send (DiffRequest, pageId, diffId) to
        latest writers;
    receive (diffs) from latest writers;
    apply diffs to page in timestamp order;
}
if (write miss) {
    create twin;
    insert page in dirtyList;
    set page status to read-write;
} else
    set page status to read-only;
```

PageRequestServer:

```
receive (PageRequest, pageId) from req
    node;
if (copyset={pid} and
    page status=read-write)
    set page status to read-only;
copyset := copyset ∪ {req};
send (page copy, copyset) to req node;
```

Barrier:

```
CreateInterval;
if (not manager) {
    lastLt := logical time of last local
        interval known at manager;
    send (Barrier, VC, local intervals
        between lastLt and VC[pid])
        to manager;
    receive (intervals);
    IncorporateIntervals(intervals);
} else {
    for each client c {
        receive (Barrier, VCc, intervals);
        IncorporateIntervals(intervals);
    }
    for each client c
        send (intervals between VCc and VC)
            to c;
}
```

Figure 3.1 -TreadMarks memory consistency algorithm.

manager's current vector time.

The storage for the diffs, the write notice records and the interval records is not freed until a garbage collection is performed, i.e. a process effectively maintains a log of all shared memory accesses since the last garbage collection. During garbage collection, all

processes synchronize at a barrier. Then each process updates its local copies of shared pages (either the copy is discarded or all diffs are requested). After this, each process sends a message to the garbage collection manager, informing it that garbage collection is complete. After receiving messages from all other

processes, the manager replies to each of them. Each process then frees all data structures used by the LRC protocol and resumes normal execution.

4. Recovery Algorithm

4.1 System Model

We consider a lazy-invalidate release consistent DSM system [Keleher 94] composed of a set of nodes strongly connected by an interconnection network. Each node executes exactly one process, which communicates with other processes by exchanging messages. Messages are transmitted reliably by using protocols specific to the operations of the DSM system. Nodes fail independently by halting and all surviving processes detect the failure within bounded time, according to the fail-stop model [Schneider 84]. Processes execute in a piecewise deterministic manner [Strom 85] and the only non-deterministic events are shared memory reads. Processes have no data races.

4.2 New Data Structures

The recovery algorithm uses the existing LRC data structures. In addition, each process maintains in its local volatile memory the following data structures:

- sentLog[NPROCS]:** array with lists of pairs $\langle VC_{acq}, VC_{rel}' \rangle$ with the vector times used by the local process *rel* to determine which set of intervals to send to the acquiring process *acq*.
- receivedLog[NPROCS]:** array with lists of pairs $\langle VC_{bef}, VC_{aft} \rangle$ with the vector times of the local process before and after incorporating the intervals received at acquire time from other processes.
- sent_to_mgrLog[NPROCS]:** array with lists of pairs $\langle lastLt_{mgr}, intLt_p \rangle$ with the last logical time of *p* known to the barrier manager *mgr* and the logical time of the last local interval record of process *p*.
- received_by_mgrLog:** list of pairs $\langle VC_{bef}, VC_{aft} \rangle$, maintained by the manager of the barrier, with the vector times of the local process before and after incorporating the intervals received at the barrier crossing.

The send logs *sentLog[i]* and *sent_to_mgrLog[i]* record information sent by the local process to process *i* and are necessary to reconstruct the state of process *i* if process *i* crashes. The receive logs *receivedLog[i]* and *received_by_mgrLog* record information received by the local process from process *i* and are necessary to reconstruct the send logs of process *i* if process *i* crashes.

4.3 Logging

This subsection describes the logging operations which were added to the base TreadMarks LRC algorithm. Figure 4.1 presents the pseudo-code for this algorithm with fault tolerance extensions in boldface. In Figure 4.1, *VC'* is the approximate vector time of each process. *VC'* is simply a vector time equal to *VC* except that *VC'[pid]* is the logical time of the last local interval record created, which may be smaller than *VC[pid]*. Note that *VC'* is implicitly updated whenever *VC* is updated or a local interval is created.

4.3.1 Logging at Acquires

During an acquire, the releasing process must log the instant at which it received the acquire request and the instant at which the acquiring process issued the request. This way, during recovery, the releaser will be able to inform the recovering process of the set of intervals received during that acquire. Therefore, the acquiring process will invalidate the same pages at the same execution point, which will cause the same read faults to occur and the same pages and diffs to be requested from the surviving processes. The diffs that will be received are the same, because the LRC protocol keeps these diffs in the memory of the processes which sent them. The pages that will be received may not be exactly the same that were received before the fault, but our algorithm guarantees that they differ only in the parts that will not be accessed by the process.

Each acquire operation is uniquely identified by the logical time of the process at the time the acquire is performed. In the base LRC protocol, incrementing the logical time is delayed until there is communication with other processes, to optimize the case when the lock is re-acquired by the same process. In the fault tolerant LRC protocol, the logical time is incremented at every acquire because during replay it is necessary to uniquely identify remote acquire operations. The logical time increment cannot be associated with interval record creation because interval creation can be asynchronous (in LockAcquireServer). During re-execution, it would not be possible to replay these asynchronous increments and therefore acquires would not be identifiable by logical time. The logical time of the process is also incremented at every release because every time a process executes a release a new logical interval begins. If an interval record is created for this new interval, its logical time must be different from the logical times of previous intervals. For the same reason, the logical time is also incremented at every barrier.

At each remote acquire, the releasing process creates a new entry in the *sentLog*, where it saves its approximate vector time, *VC_{rel}'*, and the acquirer's vector time, *VC_{acq}*. The intervals between these vector

LockAcquire:

```

VC[pid] := VC[pid] + 1;
VCbest := VC;
send (AcqRequest, VC) to lock manager;
receive (intervals)
IncorporateIntervals(intervals);
log <VCbest, VC> in receivedLog[pidacq];

```

LockAcquireServer:

```

receive (AcqRequest, VCacq) from acquirer;
wait until lock is released;
if (an interval was not created since the
    last release of this lock)
    CreateInterval;
log <VCacq, VC'> in sentLog[pidacq];
send (intervals between VCacq and VC') to
    acquirer;

```

LockRelease:

```

VC[pid] := VC[pid] + 1;
if (lock request pending)
    wakeup LockAcquireServer;

```

CreateInterval:

```

if (dirtyList is not empty) {
    -- removed logical time increment --
    insert new interval i in
        procArray[pid];
    for each page in dirtyList
        create write notice record for
            interval i;
    clear dirtyList;
}

```

IncorporateIntervals(intervals):

```

for each i in intervals {
    insert record for i in
        procArray[i.idCreat];
    VC[i.idCreat] := i.vc[i.idCreat];

    for each write notice in i {
        store write notice;
        if (twin exists for the page write
            notice refers to) {
            if (a write notice corresponding
                to the current writes was not
                already created)
                CreateInterval;
            create diff tagged with VC[pid];
            delete twin;
        }
        set page status to no-access;
    }
}

```

DiffRequestServer:

```

receive (DiffRequest, pageId, diffId)
    from req node;
if (diff is not created) {
    create diff tagged with VC[pid];
    delete twin;
    set page status to read-only;
}
send (diff) to req node;

```

PageFaultHandler:

```

if (page status = no-access) {
    if (local page copy not initialized) {
        if (first GC done) {
            send (PageRequest, pageId)
                to page manager;
            receive (page copy, copyset);
        } else
            zero-fill local page copy;
    }
    send (DiffRequest, pageId, diffId) to
        latest writers;
    receive (diffs) from latest writers;
    apply diffs to page in timestamp order;
}
if (write miss) {
    create twin;
    insert page in dirtyList;
    set page status to read-write;
} else
    set page status to read-only;

```

PageRequestServer:

```

receive (PageRequest, pageId) from req
    node;
--removed possible page status change--
if (req ≠ copyset) {
    copyset := copyset ∪ {req};
    send (page copy, copyset) to
        req node;
} else {
    send (page copy from last consistent
        checkpoint, copyset) to req node;
}

```

Barrier:

```

VC[pid] := VC[pid] + 1;
CreateInterval;
if (not manager) {
    lastLt := logical time of last local
        interval known at manager;
    VCbest := VC;
    log <lastLt, VC'[pid]> in
        sent_to_mgrLog[pidmanager];
    send (Barrier, VC, local intervals
        between lastLt and VC'[pid])
        to manager;
    receive (intervals);
    IncorporateIntervals(intervals);
    log <VCbest, VC> in
        receivedLog[pidmanager];
} else {
    VCbest := VC;
    for each client c {
        receive (Barrier, VCc, intervals);
        IncorporateIntervals(intervals);
    }
    log <VCbest, VC> in
        received_by_mgrLog;
    for each client c {
        log <VCc, VC'> in sentLog[c];
        send (intervals between VCc and VC')
            to c;
    }
}

```

Figure 4.1 -TreadMarks memory consistency algorithm with fault tolerance extensions.

times can be obtained from *procArray*. During recovery, these intervals are sent to the recovering process to replay the acquire operation. VC_{rel}' is logged instead of VC_{rel} because when the acquire request is serviced, an interval record with logical time $VC_{rel}[pid]$ might not have been created. If later such an interval record is created, it is necessary to guarantee that the corresponding write notices are not sent to the acquirer during replay of the acquire operation.

The releaser does not need to log the whole vector times; it suffices to log the positions where $VC_{rel}'[i] > VC_{acq}[i]$. We currently do not use this optimization as its benefits would be minimal for our number of processors, but it could be used to compress the log if very large vector clocks were used.

After a failure, it is also necessary to recover the *sentLog* to tolerate subsequent faults in other processes that received write notices from the recovering process. After an acquire operation, the acquiring process records in its *receivedLog* its vector times from before and after incorporating the intervals. Although this might not be an exact copy of the log entry created by the releaser, it contains information about which intervals were received, because the positions where $VC_{acq}[i] > VC_{bef}[i]$ have the same values as the positions where $VC_{rel}'[i] > VC_{acq}[i]$ in the log of the releaser. During recovery, the processes that received intervals from the recovering process send it these log entries, so that the recovering process can recover its *sentLog*.

4.3.2 Logging at Barriers

During barrier crossing, the manager logs in its *sentLog* the vector times received from each process and its approximate vector time, VC' , at the time of reply. If a non-manager process is recovering, this information allows the manager to re-send exactly the same intervals as before the failure.

Each non-manager process logs in its *sent_to_mgrLog* a tuple $\langle lastLt, intLt \rangle$ with the logical time of the last local interval that is known at the manager and the logical time of the last local interval record created. This information is used to calculate the local intervals sent to the manager, but it does not uniquely identify the barrier crossing at which they were sent. However this information is implicit in the position of the tuple in the *sent_to_mgrLog*, because barrier crossings are global operations performed by every process and barrier managers are statically assigned.

When a process receives the reply from the manager, it logs in the *receivedLog* its vector times before and after incorporating the received intervals. This is exactly the same logging operation that is performed during lock acquires. To recover the

information in the *sent_to_mgrLog* of failed processes, the manager logs in the *received_by_mgrLog* its vector times before and after incorporating the intervals at the barrier. This entry keeps the same information kept by the set of all corresponding *sent_to_mgrLog* entries at non-manager processes.

4.3.3 Page Logging

On the first access to a page, the LRC protocol requests an initial page copy from the page manager. This copy may have writes which do not have corresponding diffs, for two reasons. Since pages are initially kept writable at the manager, it may perform some writes which are not captured in diffs. Also, immediately after each GC operation, all the diffs are discarded and writes are kept only in page copies.

Suppose that some process requests a page copy and reads the value of one of these writes. Suppose later the process fails and is re-started. During recovery, the recovering process will request the same page copy. However, the manager cannot send its current copy because it may have writes that follow (according to the LRC partial order) the accesses the recovering process is about to perform. These writes possibly overwrite the values which the recovering process read during normal execution. Figure 4.2 exemplifies this problem¹. Suppose P_1 is the manager of some page and no other process requested a copy of that page. The page is writable when $w(1)3$ is performed, and hence no page fault is generated and a twin is not created. Later P_2 acquires lock i and initiates a $r(1)$ access. This will trigger a page fault and a page request is sent to P_1 . Upon receipt of the reply, P_2 completes the $r(1)3$ access and performs $w(1)1$. Later P_1 acquires lock i and performs $r(1)$. Hence the diff corresponding to $w(1)1$ will be requested from P_2 and applied to the local page copy. If P_2 fails, during recovery it will request the same page copy from P_1 . If P_1 replies with its current copy, the $r(1)$ access at P_2 will return 1, and thus P_2 will not reach the same state.

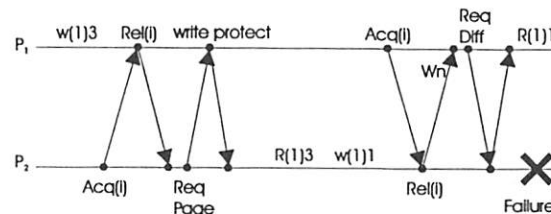


Figure 4.2 - The problem of sending current page copies during process recovery.

¹ $r(x)y$ is used to denote a read operation from address x returning value y . $w(x)y$ denotes a write of value y to address x . Wn is used to denote a set of write notices.

To avoid maintaining two copies of each shared page in the memory of page managers, we introduced the following modifications. When processes are started, all their shared pages are write protected; this includes page managers. When the first fault occurs for a page, it is initialized to all zeros. Thus all processes start from the same page contents. These modifications cause at most one additional page fault and diff to be created for each page. Until the first global garbage collection operation is executed, there is no need to issue page requests, as all writes have corresponding diffs. After the first global GC, all shared pages are saved in the consistent checkpoint of all the processes. After the first global GC, if a process receives a page request from a process which does not yet belong to the copyset of the page, it returns its current copy. If, on the other hand, the requesting process already belongs to the copyset, the manager must fetch the initial page copy from the last global checkpoint and return that copy.

With this algorithm, the page copies received during normal execution and recovery can be different, since during normal execution the pages may contain writes which do not precede the local accesses. However, this poses no problem as it is guaranteed that a process with no data races will not read these values and will therefore re-execute as before the failure.

4.3.4 Termination Related Logging

When a diff is created during asynchronous handling of a diff request, it can contain writes which were performed immediately before the request arrived. This means that recovery must end after these writes are performed, if the system is to be in a consistent state. Figure 4.3 shows an example where process P_1 acquires lock m from process P_2 . During this acquire, P_1 receives a write notice for the page where $w(0)1$ was performed. P_2 then locally acquires lock b and performs $w(2)4$ on the same page. When process P_1 requests the diff for the write notice it received, P_2 write protects the page where the writes were performed and creates the diff. The diff contains both $w(0)1$ and $w(2)4$.

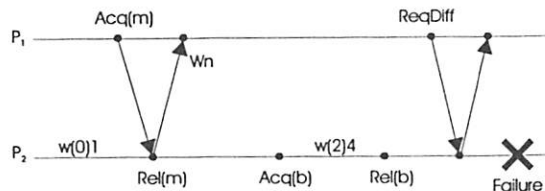


Figure 4.3 - Diff creation due to asynchronous message reception which keeps write done immediately before the request arrives.

Suppose recovery of P_2 ends at the $Rel(m)$. If process P_1 acquires lock b and performs $r(2)$, it will get the value 4, which was not written by any process.

To guarantee that recovery ends after the last diff creation, it is necessary to tag each diff with the logical time of the process where it was created. During recovery, when the logical time of the process becomes greater than the logical time of the last diff, it is guaranteed that all writes sent in diffs are performed.

Recovery must also terminate only when all intervals that were sent to other processes are finished. If this did not happen, then after recovery some processes would have intervals from the faulty process with logical times greater than the current logical time of the recovered process. These intervals would not be replayed and eventually intervals with the same logical times would be created but with possibly different write notices. This requirement does not introduce any new logging operation. Although interval record creation can be asynchronous, each interval is uniquely identified by its logical time, and during replay, the process will re-execute until the last interval sent to other processes is finished.

4.4 Checkpointing

Periodically, the processes can take independent checkpoints to optimize recovery time. When one of these checkpoints is completed, its previous independent checkpoint can be discarded.

Several optimizations can be used to speed up independent checkpointing. Initially all shared pages are excluded from the checkpoints. When there is a write fault, the corresponding page is included in the next checkpoint. When a page is invalidated, it is excluded from the next checkpoint, because its state can be recovered from the corresponding diffs. This is different from incremental checkpointing because the page may be written after one checkpoint and still need not be included in the next one. The diffs received from other processes are not included in the checkpoints because they can be easily recovered from the processes where they were created. The diffs created locally and sent to other processes also need not be included in the checkpoints because they can be recovered from the processes that received them.

During the global garbage collection operations already performed by the TreadMarks LRC protocol, a consistent global checkpoint is created, and the LRC data structures used for recovery are discarded. As processes already have to coordinate to perform GC, there is no additional coordination overhead to establish the global checkpoint. This consistent checkpointing mechanism is used to efficiently tolerate multiple faults.

If multiple faults are detected, all the processes are rolled back to the last consistent checkpoint.

The consistent checkpointing and garbage collection protocol proceeds as follows. First, all processes synchronize at a barrier. Then each process updates its local page copies, as in the normal GC protocol. Each process then sends a message to a checkpoint server requesting to save its checkpoint and waits for the reply. After having received and saved all the checkpoints, the checkpoint server sends acknowledgment messages to all processes. Each process then frees all entries of the send and receive logs (and all LRC data structures). Afterwards normal execution is resumed. This is exactly the protocol performed during LRC GC; the only additional overhead is due to checkpoint transmission and storage.

The incorporation of consistent checkpointing with garbage collection does not permit increasing the checkpointing interval. However GC operations must be relatively infrequent to not affect the performance of the DSM system. Therefore, increasing the amount of memory for the LRC data structures improves the DSM performance by reducing the frequency of both GC and consistent checkpointing.

Usual optimizations such as incremental and non-blocking checkpointing [Elnozahy 92, Li 90] can be used to reduce checkpoint overhead. The performance of copy-on-write checkpointing might be affected by the low amount of free memory at GC time. However, performing GC at a lower occupied memory threshold would solve this problem. If non-blocking checkpointing is used, single failures while the checkpoints are being saved would cause the system to rollback to the last consistent checkpoint. To avoid this, the LRC data structures needed for recovery could be freed only after the checkpoints are saved on stable storage. This would also require GC to be done at a lower occupied memory threshold, because processes would need memory to continue execution while the checkpoints are being saved.

4.5 Recovery

In order to simplify the following presentation, we do not refer to barrier crossings but only to lock acquires. Discussion of recovery for barriers is deferred until the next section.

4.5.1 Data Collection

When a process failure is detected, a new process is started from the most recent checkpoint of the failed process, p . The recovering process logically broadcasts a message informing it is in recovery. This message contains the current logical time of the process, T_{ckpt} , which is the one saved in the checkpoint file, and the

logical time of the last local interval record created, $localIntLt$. All surviving processes which have $VC[p] > localIntLt$ reply by sending their vector times and the identifiers of the diffs created by the recovering process, which are kept in their local memories.

The recovering process then requests all its intervals with a logical time greater than $localIntLt$ from the process which has the largest $VC[p]$. All diffs generated by the process after or at its current logical time are also requested from the processes that cache them. The process then builds a list of all the intervals it had created during the failure-free period. This list, ordered by descending logical time of the recovering process, is prepended to $procArray[p]$.

The recovering process also sends to each other process, c , the $VC_{acq}[c]$ of its last entry in $sentLog[c]$. Each process replies by sending its $receivedLog[p]$ entries where $VC_{bef}[c]$ is greater than the received $VC_{acq}[c]$. These entries are appended to the $sentLog[c]$ of the recovering process.

Each process also sends the identifiers of the pages for which it has a local copy and that are managed by the recovering process. This enables the recovery of the copyset information for each of these pages.

Each process, c , also sends a list of tuples $\langle VC_{acq}, VC_{rel}', intervalSet \rangle$ corresponding to its $sentLog[p]$ entries where $VC_{acq}[p] > T_{ckpt}$, where $intervalSet$ is the sequence of intervals received during that acquire. The list of the prefixes $\langle VC_{acq}, VC_{rel}' \rangle$ of these tuples is appended to the $receivedLog[c]$ of the recovering process. All the lists received are merged into a single list, $acquireList$, of tuples $\langle lt, intervalSet \rangle$, where lt is the logical time of the acquire at the recovering process. This list is ordered by ascending lt .

4.5.2 Execution Replay

After the data collection phase described above, the process starts to re-execute. During execution replay, requests from other processes are blocked and the page fault handler is the same as for the normal execution.

During the first phase of recovery, each remote acquire operation is replayed. The next remote acquire is identified by the logical time of the next entry in $acquireList$. The acquire replay is the same as the normal acquire, except that no local intervals are created since they were already recovered. The intervals from other processes are prepended to the $procArray$ lists corresponding to the processes which created them.

There is however a case when the local data structures created during an acquire operation are affected by asynchronous message handling and these data structures are not recoverable from other processes. Figure 4.4 shows an example of this, when a diff is created during an acquire operation where a write

notice is received for a writable page. In Figure 4.4, all writes are performed on the same page.

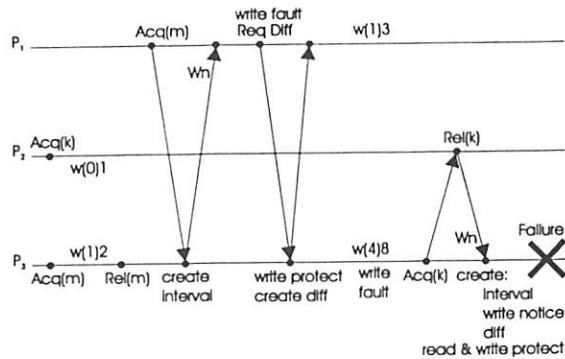


Figure 4.4 - Creation of a diff which is not sent to other processes, during an acquire operation.

If process P_3 fails and is re-started, the asynchronous reception of the lock and diff requests, sent by P_1 , cannot be replayed. If the diff created during the last acquire of P_3 is not later sent to other processes, it must be recovered during replay of the acquire operation. The diff created during recovery cannot have all the writes that were performed since the last write invalidate. For instance, in the example of Figure 4.4, $w(1)2$ and $w(4)8$ cannot both belong to the diff for the write notice created in the last acquire, because $w(1)2$ precedes $w(1)3$ while $w(4)8$ is concurrent with $w(1)3$.

To solve this problem, when a process replays an acquire operation, a diff is created if a received interval, r , has a write notice for a writable page, and there is a local interval, i , with a write notice with no diff for the same page, such that $i.vc[r.idCreat] < r.vc[r.idCreat]$.

To create the diff, all diffs that correspond to write notices for the same page, which belong to intervals that follow the last acquire where the page was invalidated, are applied to the current twin. Only then is the new diff between the page and its twin calculated. This diff is associated with the write notice without diff. During acquire operations where it is not necessary to create a diff, the page twin is simply discarded.

The first phase of recovery proceeds until one of these situations occurs:

- There are no more entries in *acquireList*, and the logical time of the recovering process, $VC[p]$, becomes greater than both the logical time of the first interval in *procArray[p]* (the last interval that was sent to another process) and the logical time of creation of the last diff created by p during the failure-free execution.
- The logical time of the recovering process, $VC[p]$, is greater than the logical time of the first interval in *procArray[p]* and the process is about to re-execute the next acquire in *acquireList*.

When the first phase of recovery ends, the recovering process applies to the twin of each writable page all the diffs corresponding to write notices for the page which belong to intervals that follow the last acquire where a write notice was received for that page, and whose logical time of creation is smaller than the current logical time. If the last of these write notices does not have a diff or if the twin does not become equal to the page, it keeps the page writable. Otherwise it write invalidates the page. The *dirtyList* keeps only the writable pages whose last write notice has a diff or does not exist. This page state recovery procedure recovers the state of shared pages which can be affected by asynchronous handling of diff requests, which can in turn correspond to write notices for intervals created during asynchronous handling of acquire requests.

If after the first phase of recovery there are no more entries in *acquireList*, recovery terminates. Otherwise, replay of remote acquire operations must continue. However, after this phase, the intervals created were not sent to other processes and therefore acquire operations are executed as during normal execution. This is another situation where a LRC protocol data structure is created but is not transmitted to another process. To recreate these data structures the following procedure is used. Immediately before each of the subsequent remote acquire operations, but after having incremented the logical time of the process for that acquire, the recovering process executes the same page state recovery procedure as was executed at the end of the first phase of recovery. Next, the acquire operation is re-executed as during the failure-free period, creating local intervals if needed. Re-execution continues until there are no more entries in *acquireList*. At this point, if the logical time of the recovering process is greater than the logical time of creation of the last diff created during the failure-free execution, recovery terminates. Otherwise re-execution continues until the logical time becomes greater than the time of creation of the last diff. The recovering process then executes the same page state recovery procedure as was executed at the end of the first phase of recovery, after which recovery terminates. At this point, all intervals that were sent during failure-free operation are created and all writes sent in diffs are performed.

One straightforward extension to the algorithm, to allow faster recovery, consists of logging which diffs and pages were requested by each process. This allows the recovering process to prefetch all the diffs and pages needed for recovery. This would provide an interesting benefit. As the processes would not need to communicate with other processes during recovery except for the initial information gathering phase, immediately after this phase other failures could be tolerated.

4.6 Recovery for Barriers

For barrier crossings at barriers that are not managed by the recovering process, recovery proceeds exactly as for lock acquires. There are, however, some differences in the replay of barrier crossings when the process is the manager of the barrier. To replay these operations, when a process p fails, it sends the number n of entries in its *received_by_mgrLog* in its initial broadcast message. Each process then sends a list² of tuples $\langle intervalSet \rangle$ corresponding to its intervals between the logical times in its $\langle lastLt, intLt \rangle$ entries in *sent_to_mgrLog* $[p]_m$ where $m > n$.

The manager then builds a list, *barrierList*, where each *barrierList* $_i$ entry is an interval set calculated as the union of the i^{th} entries in all the lists of intervals sent by each other process. Each entry in *barrierList* keeps all the intervals received by the manager at the crossing of one of the barriers it manages. Therefore these entries are used to supply the same write notices to the manager during replay of these crossings. The *received_by_mgrLog* is recovered incrementally as each barrier crossing is replayed.

To recover its *sent_to_mgrLog*, the process also sends in its initial broadcast message the number of entries, k_q , in each *sent_to_mgrLog* $[q]$ list. Each process, q , then selects from all its *received_by_mgrLog* $_i$ entries, where $i > k_q$, the logical times corresponding to the recovering process in the pair $\langle VC_{bef}, VC_{aft} \rangle$. It then sends to the recovering process a list of all the selected $\langle lastLt, intLt \rangle$ pairs. This list is then appended to the *sent_to_mgrLog* $[q]$ list at the recovering process.

5. Experimental Results

5.1 Implementation

We implemented the logging part of our algorithm on top of TreadMarks as described in section 4.3; and we used the libckpt library [Plank 95] to create the process checkpoints. This library implements incremental and non-blocking checkpointing with copy-on-write. However, incremental checkpointing was disabled in our experiments because it write protects all data pages after each checkpoint, and this would interfere with TreadMarks' management of page protections. Currently, pages which are never invalidated are written in every checkpoint. We used the facilities provided by libckpt to include and exclude memory from the checkpoints, to manage the portion of the shared address space which is checkpointed.

The recovery part of our algorithm is partially implemented. At the end of a run, all processes wait for the re-execution of one process. This process rolls back to the start of the application, gathers the recovery information needed to build the *acquireList* (section 4.5.1), and re-executes its portion of the computation.

In order to evaluate our algorithm, we compare it with the logging algorithm proposed in [Suri 95] for LRC based systems. We implemented the logging algorithm of [Suri 95] on top of TreadMarks. This algorithm logs references to the diffs obtained from other processes and copies of the messages received at lock acquires and barrier crossings. When local diffs or intervals are transmitted to other processes, the log is synchronously flushed to a local disk in each workstation.

5.2 Experimental Setup

Our measurements were performed on a network of SPARCstations 10/30 running SunOS 4.1.3, connected by a 10Mbps Ethernet network. Each SPARCstation has a V8 SuperSPARC processor running at 36MHz, 36 Kbytes of internal cache and 32 Mbytes of memory. Checkpoints were written to a Seagate Barracuda disk via NFS. Our results are based on the execution of three parallel applications included in the TreadMarks distribution:

SOR: An implementation of the red-black successive over-relaxation algorithm [Keleher 94].

TSP: Solves the traveling salesperson problem, using a branch and bound algorithm [Carter 91].

Water: An N-body molecular dynamics simulation from the SPLASH benchmark suite [Singh 91], which calculates forces and potentials in a system of water molecules in liquid state.

We ran two sets of experiments. First, a series of experiments was performed to evaluate the overheads of our logging algorithm and the one presented in [Suri 95] when the number of processes running the application is increased. For this first set of benchmarks, SOR was run for 318 iterations on a 1024x1024 floating point matrix; TSP was run with a graph of 20 cities; and Water was run for 15 steps on 500 molecules.

The second set of experiments measures our logging algorithm in 4-processor runs, with and without checkpoint creation. For the second set of experiments, SOR was run for 1400 iterations on a 1278x2048 floating point matrix; TSP was run with a graph of 22 cities; and Water was run for 23 steps on 1472 molecules. No global garbage collections or consistent checkpoints were performed during the experiments.

² The i^{th} element in list L is denoted by L_i .

In each series of experiments, the logging algorithms are compared to the TreadMarks version modified to not issue page requests before the first GC operation. We found that this modification had a very small positive impact on the performance of applications, because it reduces the amount of data transferred. Therefore, we chose to isolate this effect from the logging overhead.

5.3 Results

In the following figures, VC Logging (Vector Clock Logging) is used to label the results of our logging algorithm, while Message Logging is used to label the results of our implementation of the logging algorithm proposed in [Suri 95].

Figures 5.1, 5.2 and 5.3 show the running time for our three test applications in the first set of experiments; and Figure 5.4 shows the log sizes for each application in 4-processor runs. As expected, adding our logging algorithm to TreadMarks almost does not affect application execution time. The graphs show that the logging mechanism introduces almost zero time overhead. This happens because it does not send extra messages, does not access stable storage, and logs only a small amount of data. Figure 5.4 shows that the space overhead of our algorithm is also very small.

On the other hand, Message Logging introduces a significant execution time overhead, and the overhead grows with the number of processes. The main causes of this overhead are the accesses to stable storage and the large amount of data logged. The Message Logging approach flushes the log to disk with a synchronous write every time local diffs or intervals are sent to another process. The Message Logging approach logs diffs and write notices, whereas the VC Logging approach only logs pairs of vector times. Therefore, the Message Logging approach has significantly larger logs. Note that the Message Logging approach keeps most of the log on disk; only the portion of the log created since the last flush to disk is kept in memory.

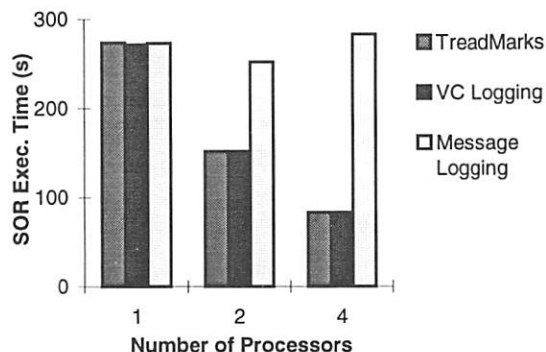


Figure 5.1 - Execution times for SOR.

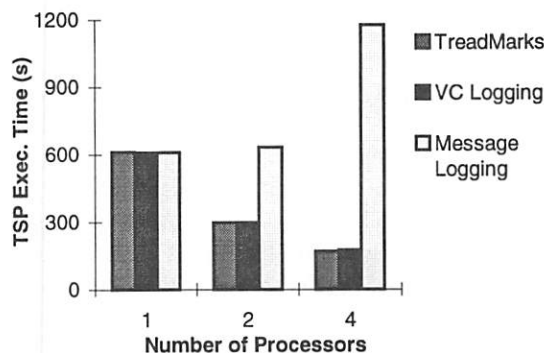


Figure 5.2 - Execution times for TSP.

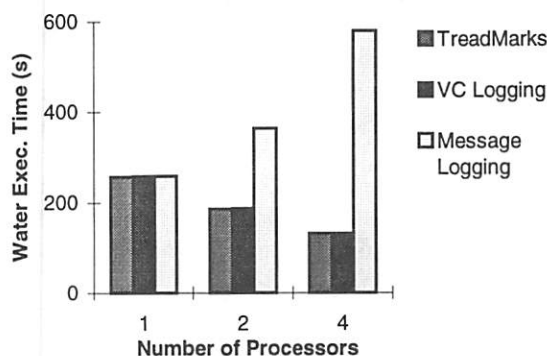


Figure 5.3 - Execution times for Water.

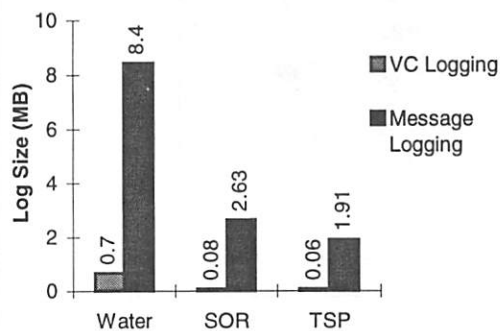


Figure 5.4 - Log sizes for first set of benchmarks.

For VC Logging, the log sizes of the applications are dictated by the number of remote synchronization operations, i.e. the number of times a lock is acquired from a remote process and the number of barrier crossings. Water uses significantly more log space than the other two applications, because it has an order of magnitude more remote synchronization operations. For Water, the *sentLog* and *receivedLog* keep 22073 VC pairs each, for a total of 353168 bytes each. For TSP, 1981 VC pairs are kept in *sentLog* and *receivedLog*. For Water and TSP most of the synchronization is done using locks, and therefore the log components related to logging at barrier managers are not significant. These components are more significant in SOR because most of the synchronization is done using barriers. SOR

keeps 1914 VC pairs in *sentLog* and *receivedLog*, 1914 logical time pairs in *sent_to_mgrLog*, and 638 VC pairs in *received_by_mgrLog*.

Figure 5.5 shows the running time of our three test applications in 4-processor runs in the second set of experiments. Six checkpoints were created for each process, during the execution of Water and TSP. For SOR, only five checkpoints were created. Log sizes and average checkpoint sizes for these runs are presented in Table 5.1. These results confirm that our logging algorithm has near zero time overhead and small space overhead.

The overhead of checkpointing is below 2% for Water and TSP. For SOR, the checkpointing overhead is approximately 22%, because the average checkpoint size in SOR is much larger than for the other two applications (as shown in Table 5.1). SOR has a larger checkpoint size because each process touches significantly more memory than in the other two applications. This inefficiency can be reduced by integrating incremental checkpointing with the LRC protocol and using modern switched network technologies.

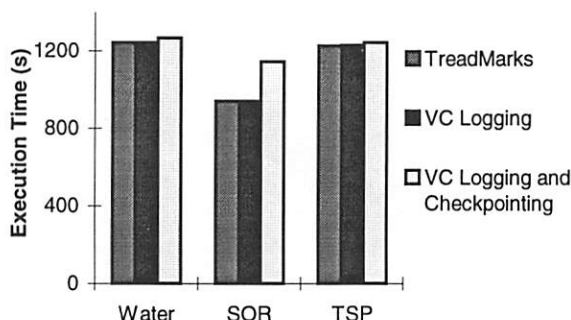


Figure 5.5 - Execution times for the second set of benchmarks.

Application	Log Size (MB)	Average Checkpoint Size (MB)
Water	3.10	3.05
SOR	0.33	7.84
TSP	0.05	2.49

Table 5.1 - Log sizes and average checkpoint sizes for benchmarks of Figure 5.5.

Figure 5.6 shows the breakdown of normal execution and recovery times for the first set of benchmarks, in 4-processor runs. Re-execution time is 72% of normal execution time for Water, 75% for SOR and 95% for TSP.

Two effects combine to make recovery faster than normal execution. First, idle time is reduced. Idle time results from waiting for locks and barriers and from remote communication latency. During recovery, the first component is completely eliminated and the second is reduced, because the recovering process is the only

one using the network. Second, Unix overhead is reduced, because the number of messages sent and received is lower ([Keleher 94] shows that at least 80% of the kernel execution time is spent in the communications routines).

Water has a high remote synchronization rate, resulting in a large amount of idle time during normal execution. During recovery, idle time resulting from this activity is greatly reduced. SOR has a lower synchronization rate, but processes synchronize with barriers and initiate data transfers at approximately the same times, which results in idle time waiting for network availability. TSP has a very small amount of idle time, and therefore its recovery time is very close to its normal execution time.

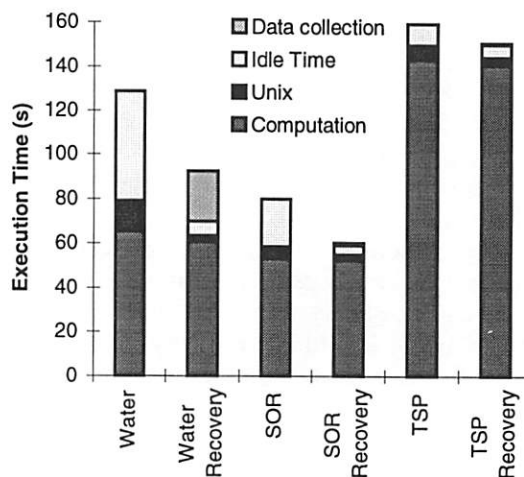


Figure 5.6 - Execution time breakdown for normal execution and recovery of the first set of benchmarks.

6. Conclusions

This paper presented a lightweight logging algorithm for lazy release consistent distributed shared memory. This algorithm handles recovery from single node failures, one of the most common failure scenarios in networks of workstations. Furthermore, the algorithm is integrated with a consistent checkpointing scheme, which allows multiple failures to be tolerated.

The logging algorithm is very efficient, because it is tightly integrated with the LRC memory coherence protocol. Our experiments show that the logging overhead for recovery of lazy release consistent DSM is very low, both in time and space.

Acknowledgments

The authors would like to thank the anonymous reviewers, Willy Zwaenepoel, Pete Keleher and especially our shepherd David Johnson for the comments which helped to improve the contents of this paper.

References

- [Anderson 95] T. E. Anderson, D. E. Culler and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.
- [Cabillic 95] G. Cabillic, G. Muller and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. *Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995.
- [Carter 91] J. B. Carter, J. K. Bennett and W. Zwaenepoel. Implementation and performance of Munin. *Proceedings of the 13th Symposium on Operating Systems Principles*, pp. 152-164, October 1991.
- [Castro 96] M. Castro, P. Guedes, M. Sequeira and M. Costa. Efficient and Flexible Object Sharing. *Proceedings of the 25th International Conference on Parallel Processing*, August 1996.
- [Chandy 85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.
- [Elnozahy 92] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel. The Performance of Consistent Checkpointing. *Eleventh IEEE Symposium on Reliable Distributed Systems*, pp. 39-47, October 1992.
- [Feeley 94] M. J. Feeley, J. S. Chase, V. R. Narasayya and H. M. Levy. Integrating Coherency and Recoverability in Distributed Systems. *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.
- [Janakiraman 94] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 42-51, October 1994.
- [Janssens 93] B. Janssens and W. K. Fuchs. Relaxing Consistency in Recoverable Distributed Shared Memory. *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 155-163, June 1993.
- [Janssens 94] B. Janssens and W. K. Fuchs. Reducing Interprocess Dependence in Recoverable Distributed Shared Memory. *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 34-41, October 1994.
- [Johnson 87] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 14-19, July 1987.
- [Johnson 89] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University, December 1989.
- [Keleher 92] P. Keleher, A. Cox and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Proceedings of the 19th Annual Symposium on Computer Architecture*, pp. 13-21, May 1992.
- [Keleher 94] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 Winter USENIX Conference*, January 1994.
- [Kermarrec 95] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin and I. Puaut. A recoverable distributed shared memory integrating coherency and recoverability. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, Pasadena, CA, June 1995.
- [Koo 87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23-31, January 1987.
- [Li 86] K. Li. *Shared Virtual Memory on Loosely Coupled Microprocessors*. PhD thesis, Yale University, September 1986.
- [Li 90] K. Li, J. F. Naughton and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pp. 79-88, March 1990.
- [Neves 94] N. Neves, M. Castro and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Computing*, pp. 121-129, August 1994.
- [Plank 95] J.S. Plank, M. Beck, G. Kingsley and K. Li. Libckpt: Transparent Checkpointing under Unix. *Proceedings of the USENIX Winter 1995 Technical Conference*. January 1995.
- [Richard III 93] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pp. 86-95, October 1993.
- [Schneider 84] F. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145-154, May 1984.
- [Singh 91] J. Singh, W. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.
- [Strom 85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204-226, August 1985.
- [Stumm 90] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory Algorithms. *Proceedings of the 2nd Symposium on Parallel and Distributed Processing*, pp. 719-724, December 1990.
- [Suri 95] G. Suri, B. Janssens and W. K. Fuchs. Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 279-288, June 1995.
- [Vaidya 95] N. H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 64-73, May 1995.
- [Wu 90] K. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 460-469, April 1990.

Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems

Yuanyuan Zhou, Liviu Iftode and Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

This paper investigates the performance of shared virtual memory protocols on large-scale multicomputers. Using experiments on a 64-node Paragon, we show that the traditional Lazy Release Consistency (LRC) protocol does not scale well, because of the large number of messages it requires, the large amount of memory it consumes for protocol overhead data, and because of the difficulty of garbage collecting that data.

To achieve more scalable performance, we introduce and evaluate two new protocols. The first, Home-based LRC (HLRC), is based on the Automatic Update Release Consistency (AURC) protocol. Like AURC, HLRC maintains a home for each page to which all updates are propagated and from which all copies are derived. Unlike AURC, HLRC requires no specialized hardware support. We find that the use of homes provides substantial improvements in performance and scalability over LRC.

Our second protocol, called Overlapped Home-based LRC (OHLRC), takes advantage of the communication processor found on each node of the Paragon to offload some of the protocol overhead of HLRC from the critical path followed by the compute processor. We find that OHLRC provides modest improvements over HLRC. We also apply overlapping to the base LRC protocol, with similar results.

Our experiments were done using five of the Splash-2 benchmarks. We report overall execution times, as well as detailed breakdowns of elapsed time, message traffic, and memory use for each of the protocols.

1 Introduction

Shared memory is considered an attractive paradigm because it provides a simple yet effective parallel programming model. Research in the last decade shows that it is difficult to build or provide shared memory

on a large-scale system. Although the hardware approach to implementing cache coherence has been shown to perform quite well, it requires a high engineering cost [25]. Shared virtual memory (SVM) [26], on the other hand, is a cost-effective method to provide the shared memory abstraction on a network of computers since it requires no special hardware support. The main problem with this approach has been its lack of scalable performance when compared with hardware cache coherence. The challenge is to reduce the overhead of the software coherence protocols and to implement efficient shared virtual memory that performs well with various applications on large-scale machines.

There are several factors that limit the performance of a shared virtual memory implementation. First, the large coherence granularity imposed by the underlying virtual memory page size induces false sharing and fragmentation for most applications. This effect contributes to a higher communication to computation ratio than that of the hardware cache coherence approach. The unit cost of communication of a shared virtual memory implementation is often higher than that of a hardware implementation. Second, synchronization primitives are relatively slow because they are implemented through explicit, synchronous messages. Third, performing the memory coherence protocol in software is expensive because the computation has to be interrupted to service both local and remote memory coherence protocol requests.

The known software approach to reducing the overhead of shared virtual memory is to employ relaxed memory consistency models such as Release Consistency (RC) and Lazy Release Consistency (LRC) [8, 22]. Both protocols allow multiple writers while avoiding false sharing, and both reduce overheads by maintaining coherence only at acquire and release synchronization points. The two differ on how eagerly they reconcile updates: the RC protocol propagates updates on release, whereas the LRC protocol postpones update propagations until the next acquire. Although previous prototypes have shown reasonably good performance for some applications on small systems [21], protocol overhead becomes substantial on large-scale systems. Our own experience shows that many applications do not speedup well using standard LRC-based shared virtual memory on a 32-node machine, and that the speedup curves go down when increasing the number of nodes to 64.

Recently, Automatic Update Release Consistency (AURC) [15] has been proposed as an alternative LRC-based protocol. The two protocols, AURC and LRC, have different update detection schemes. AURC uses a hardware mechanism called automatic update for update detection while LRC detects the updates in software using *diffs*. Further, in AURC updates are incorporated as they are produced into a master copy of the shared page whose (fixed) location is called its *home*. This makes update resolution in AURC extremely simple: a single full-page fetch from the home. On the other hand, in standard LRC collections of updates (*diffs*) are distributed and homeless, making update resolution more difficult to perform. Extensive performance evaluation [16] has shown that AURC outperforms standard LRC in most cases.

In this paper we propose two new home-based LRC protocols. The first, Home-based LRC (HLRC), is similar to the Automatic Update Release Consistency (AURC) protocol. Like AURC, HLRC maintains a home for each page to which all updates are propagated and from which all copies are derived. Unlike AURC, HLRC requires no specialized hardware support, using *diffs* as in LRC for update detection and propagation. This idea of using a home-based approach to build an all-software protocol similar to AURC was introduced in [17]. Our second protocol, called Overlapped Home-based LRC (OHLRC), takes advantage of the communication processor found on each node of the Paragon to offload some of the protocol overhead of HLRC from the critical path followed by the compute processor.

To evaluate the performance implications of home-based LRC, we have implemented four shared virtual memory protocols on a 64-node Intel Paragon multi-computer: HLRC, OHLRC, the standard LRC and an overlapped LRC. We compare the performance of the two home-based protocols with the two homeless LRC protocols using several Splash-2 benchmark programs. Our results show that the home-based protocols provide substantial improvements in performance and scalability over the homeless ones and that protocol overlapping using a communication processor further adds only modest improvements. By studying detailed time breakdowns, communication traffic, and memory requirements, we show also that the home-based protocols scale better than the homeless ones.

2 LRC, AURC, and Home-based LRC

In this section we briefly review the standard Lazy Release Consistency [22] and Automatic Update Release Consistency [15] protocols. We then describe our home-based and overlapped protocol variations.

2.1 Lazy Release Consistency

The standard LRC [22] is an all-software, page-based, multiple-writer protocol. It has been implemented and evaluated in the TreadMarks system [21]. The LRC protocol postpones updates to shared pages and uses the causal orders to obtain up-to-date versions. The main idea of the protocol is to use *timestamps*, or intervals,

to establish the happen-before ordering between causal-related events. Local intervals are delimited by synchronization events. In our implementation an interval on processor P is delimited by one of the following two events: (i) processor P performs a remote acquire operation, or (ii) processor P receives a remote lock request.

Every writer locally records the changes it makes to every shared page during each interval. When a processor first writes a page within a new interval, it saves a copy of the page, called a *twin*, before writing to it. When the interval ends, the processor saves the interval and the page numbers that were updated in a record called a *write-notice*. The processor then compares the dirty copy of the page with the twin to detect updates and records these in a structure called a *diff*. The LRC protocol creates *diffs* either eagerly, at the end of each interval, or lazily, on demand.

On an acquire, the requesting processor invalidates all pages according to the write-notices received. The first access to an invalidated page causes a page fault. The page fault handler collects all the *diffs* for the page and applies them locally in the proper causal order to reconstitute the coherent page. Figure 1(a) shows how the protocol works with a simple example.

Such a multiple-writer protocol has several benefits. Since multiple nodes can update the same page simultaneously, the protocol can greatly reduce the protocol overhead due to false sharing. By delaying coherence actions until a synchronization point, the protocol can reduce the number of messages for protocol and data, and hence reduce software protocol overhead. Furthermore, the protocol can reduce the communication traffic due to data transfer: instead of transferring the whole page each time, the protocol transfer *diffs* to propagate updates.

On the other hand, *diff* processing can be expensive. First, the *diff* creations and applications all have substantial software overhead and pollute the processor's caches. Second, the acquiring processor may have to visit more than one processor to obtain *diffs* when multiple nodes update the same page simultaneously. Third, even when consecutive *diffs* of the same page (from multiple synchronization intervals) are obtained from one place, they have to be obtained as separate *diffs* and applied individually by the faulting processor. However, multiple *diff* traffic and application is avoided when the page has only one writer by transferring the whole page instead.

The data structures of this protocol can consume a substantial amount of memory. The memory required to store *diffs* and write notices can grow quickly since they cannot be discarded as long as there are nodes that may still need them. When implementing the protocol on a large-scale machine, memory consumption can become a severe problem. To reduce memory consumption the shared virtual memory system must perform garbage collection frequently [21].

2.2 Automatic Update Release Consistency

The AURC protocol [16] implements Lazy Release Consistency without using any *diff* operations by taking

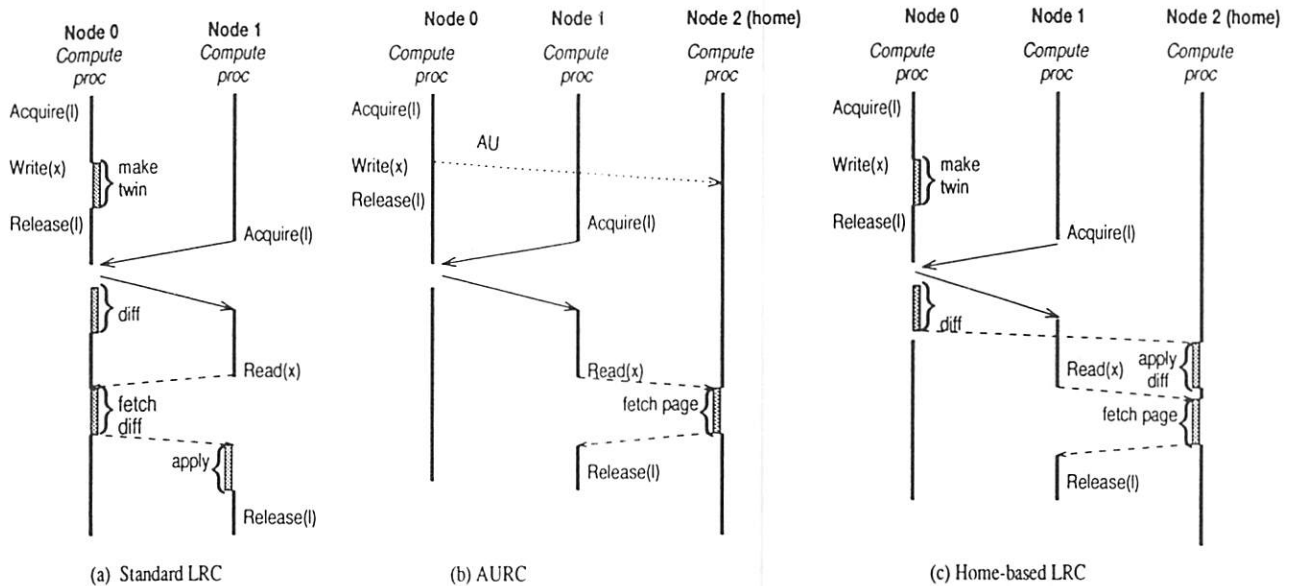


Figure 1: An example of LRC, AURC and Home-based LRC.

advantage of the SHRIMP multicomputer's automatic update hardware mechanism [5, 6]. Automatic update provides write-through communication between local memory and remote memory with zero software overhead. Writes to local memory are snooped off the memory bus, propagated, and performed on remote memory by the virtual memory-mapped network interface. AURC uses the automatic update hardware instead of diffs to detect, propagate, and merge the writes performed by multiple writers on different copies of the same page.

The basic idea of AURC is to have a *home* copy for every shared page and to set automatic update mappings such that writes to other copies of the page are automatically propagated to the home. In this way the home copy is always kept up-to-date while the other copies will be updated on demand by fetching the home copy based on a LRC-like invalidation scheme. Vector timestamps are used to ensure memory coherence before pages are either accessed or fetched. Figure 1(b) shows how AURC works with a simple example. Further details can be found in [15, 16].

The AURC protocol has several advantages over the standard LRC protocol. First, it uses no diff operations. Second, there can be no page faults when accessing home node pages, so it can reduce the number of page faults if homes are chosen intelligently. Third, the non-home nodes can bring their pages up-to-date with a single round-trip communication (with the home node). Fourth, protocol data in memory and protocol messages (write-notices) under AURC are much smaller than under standard LRC.

On the other hand, AURC usually generates higher memory bus and communication traffic than LRC, both because of the write-through nature of the automatic update mechanism and because page misses always require whole-page transfers. It is nevertheless true that the latter can be outstripped by larger aggregate diff sizes in LRC. Overall, the major tradeoff between AURC

and LRC is between bandwidth and protocol overhead.

In addition, the AURC method requires an automatic update mechanism. It cannot be implemented on a machine without the automatic update hardware support.

2.3 A Home-based LRC Protocol

We propose a new protocol called Home-based LRC (HLRC). Similar to the AURC protocol, it uses a "home" node for each shared page to collect updates from multiple writers. Unlike the AURC method, it requires no automatic update hardware support; it can be implemented on workstation clusters or multicomputers with traditional network interfaces.

The main idea in the HLRC protocol is to compute diffs at the end of an interval to detect updates and to transfer the updates as diffs to their homes. The lifetime of diffs is extremely short, both on the writer nodes and the home nodes. Writers can discard their diffs as soon as they are sent. Home nodes apply arriving diffs to their copies as soon as they arrive, and can then discard them. Later, on a page fault following a coherence invalidation, the faulting node fetches the whole page from the home node. Figure 1(c) shows an example of how HLRC works.

HLRC inherits most of the advantages of AURC over standard LRC: accesses to pages on their home nodes cause no page faults, non-home nodes can bring their shared pages up-to-date with a single round-trip message, and protocol data and messages are much smaller than under standard LRC.

There are two main differences between HLRC and AURC. The first is that the HLRC protocol delays propagating updates until the end of an interval, whereas the AURC method uses the automatic update mechanism to propagate eagerly. The second difference is that the HLRC protocol merges multiple updates to the same page as diffs in software and sends them together in a

single message, whereas the AURC method may send one or more messages (depending on the behavior of the hardware update combining mechanism). Thus, the HLRC method pays diffing overhead to detect updates and a regular message for each page, whereas the AURC method pays zero software overhead on update detection or message initiation.

2.4 Overlapped Protocols

Many parallel architectures [27, 14, 24, 29] contain dedicated communication and/or protocol processors that take over most of the overhead of performing these operations from the compute processor(s). Even though the occupancy of these processors is in general low, they cannot be used for general application computation since they are running a server polling loop in kernel mode. In addition, in most cases, the communication processor is one generation behind the compute processors.

This section describes two protocol variations, called Overlapped LRC and Overlapped Home-based LRC, that extend the use of the communication co-processors on the Paragon as protocol processors to overlap some of the SVM protocol overhead with computation.

2.4.1 Overlapped LRC

The Overlapped LRC (OLRC) protocol uses the communication co-processor for two tasks: to perform diffs and to service remote fetch requests. For the first task the compute processor asks its co-processor to compute diffs at the end of each interval for all pages that have been updated during that interval. Once performed, diffs are stored and propagated to remote nodes on demand.

The second task of the co-processor is to service remote requests for diffs and pages without interrupting the compute processor. These requests are sent by remote processors on a page access fault in order to bring their copies up-to-date according to the information previously provided by the write-notices. The co-processor sends the requested diffs if available. If a diff computation is in progress, the co-processor queues the request until the diff is ready. Full-page requests are issued only when the faulting node does not have a local copy for the page.

Other operations, like twin creation and diff application, are still performed by the compute processor. Usually, these do not expose enough overlapping potential to justify co-processor involvement. Also, the remote lock acquire operation, which requires coherence control handling, is still serviced by the main processor. We made this decision in an early stage of the implementation to keep the co-processor interface independent of any particular protocol.

Figure 2(a) shows an example of how the OLRC protocol overlaps its overhead with computation. Before performing `write(x)` on node 0, the compute processor creates a twin for the page holding `x`. Node 0 then releases the lock. Later, when node 1 tries to acquire lock `l`, it sends a lock request to node 0. After servicing the request, the compute processor of node 0 asks its co-processor to compute diffs. After acquiring the lock, the compute processor of node 1 invalidates the page

holding `x` and continues its computation. The `read(x)` on node 1 causes a page fault, which triggers a diff request sent to node 0. The co-processor of node 0 handles the request. Finally, the compute processor of node 1 receives the diff, applies it to its local copy, and continues its computation.

The OLRC approach is similar to the standard LRC protocol. Moving the diff computation and fetching to the communication co-processor is easy. Applications could benefit from overlapping these operations with computation. At the same time, OLRC has the same drawbacks as LRC. The overhead of diff application as well as memory consumption can greatly affect application performance.

2.4.2 Overlapped HLRC

The Overlapped HLRC (OHLRC) uses the communication co-processor to perform three operations:

- Compute diffs after each interval and send them to their home node.
- Apply diffs on the local copy (at the home node).
- Service the remote requests for pages at home.

After completing a diff of a given page, the co-processor sends it to the co-processor of the home of that page. There the diff is applied and the timestamps of the corresponding page are updated. When servicing a page fetch remote request, the co-processor compares the timestamps in the request with the local timestamps of that page to ensure that the required updates are in place. If an element in the local timestamp vector is smaller than the corresponding element in the timestamp vector of the fetch request, then some diffs are still in progress and the page request is put into the pending request list attached to that page. Once all the necessary diffs have been applied, the co-processor sends the page to the faulting node.

Figure 2(b) shows an example of the OHLRC protocol. To service remote acquire requests on node 0, the current interval is ended, the co-processor starts to compute a diff, and a reply is sent back immediately to node 1, with the corresponding write notices. Node 1 invalidates the page holding `x` and then continues its computation. In the meantime, the co-processor of node 0 computes diffs and sends them to the appropriate home (node 2). The co-processor of node 2 receives the diffs and applies them to its local page. The `Read(x)` operation on node 1 causes a page fault that triggers a request sent to the home node (node 2). The co-processor of node 2 services the page request after checking the timestamps.

OLRC and OHLRC are both overlapped protocols, but their degrees of overlapping are different. Both protocols overlap diff computation and fetching with computation, but OHLRC also overlaps diff applications by performing them eagerly at the home. It appears that this may cause OHLRC to transfer more data than OLRC since OHLRC always fetches full pages from home nodes and OLRC fetches diffs instead. But this is not always true, as shown in our experiments

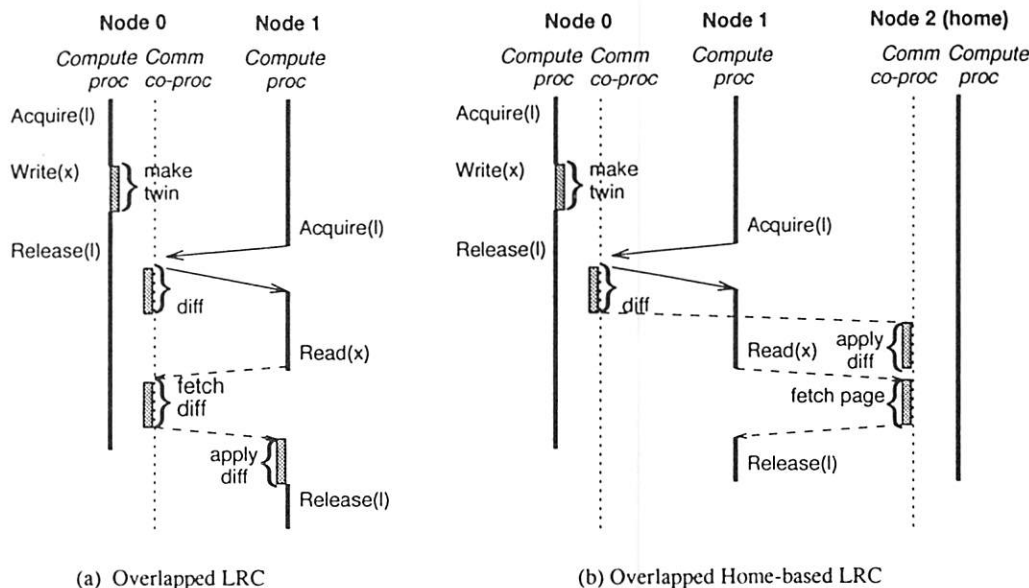


Figure 2: An example of Overlapped LRC and Overlapped Home-based LRC.

on communication traffic. Our experiments show that OHLRC is the best protocol among the four protocols.

3 Prototype Implementations

To evaluate the performance of the protocols we have implemented four SVM prototypes on a 64-node Intel Paragon multicomputer: the standard LRC protocol, the Home-based LRC (HLRC), the Overlapped LRC (OLRC) protocol and the Overlapped Home-based LRC (OHLRC) protocol. Standard LRC is our baseline implementation. It runs on the compute processor without any overlapping. Both OLRC and OHLRC use the communication processor to overlap some protocol tasks with computation.

3.1 The Paragon System

The Intel Paragon multicomputer used in our implementation consists of 64 nodes for computation. Each node has one compute processor and a communication co-processor, sharing 64 Mbytes of local memory. Both processors are 50 MHz i860 microprocessors with 16 Kbytes of data cache and 16 Kbytes of instruction cache [18]. The data caches are coherent between the two processors. The memory bus provides a peak bandwidth of 400 Mbytes/sec. The nodes are interconnected with a wormhole routed 2-D mesh network whose peak bandwidth is 200 Mbytes/sec per link [31].

The operating system is a micro-kernel based version of OSF/1 with multicomputer extensions for a parallel programming model and the NX/2 message passing primitives. The co-processor runs exclusively in kernel mode, and it is dedicated to communication. The one-way message-passing latency of a 4-byte NX/2 message on the Paragon is about 50 μ sec [27]. The transfer bandwidth for large messages depends on data

alignment. When data are aligned properly, the peak achievable bandwidth at the user level is 175 Mbytes/sec. Without proper alignment, the peak bandwidth is about 45 Mbytes/sec.

The operating system uses an 8 Kbyte page size for its virtual memory, though the hardware virtual memory page size is 4 Kbytes. All implementations use the `vm_protect` Mach system call to set access protection for shared pages. Access faults can be handled by either the external memory manager or the exception handling mechanism. We used the Mach exception handling mechanism for efficiency reasons.

In our prototypes for the overlapped protocols we extended the functionality of the communication co-processor with SVM protocol related operations. The standard LRC protocol is implemented exclusively at user-level using the NX/2 message library. For the overlapped protocols we modified the co-processor kernel to perform diff-related operations.

3.2 Shared Memory API

All four prototypes support the programming interface used with the Splash-2 [28] benchmark suite. This is different from the APIs supported by other software shared virtual memory systems, such as TreadMarks. The main rationale for our decision to implement the Splash-2 API is to allow programs written for a release-consistent, shared-memory multiprocessor to run on our systems without any modification.

In our implementations, all virtual address space can be shared, and global shared memory can be dynamically allocated using `G_MALLOC`. A typical program on P processors starts one process first that allocates and initializes global data and then spawns the other P-1 processes. All P processes perform the computation in parallel and join at the end. The only synchronization primitives used in the programs are `LOCK`, `UNLOCK` and

3.3 Co-processor Interface

The communication co-processor communicates with the compute processor via cache-coherent, shared memory. For each message passing client, which can be a kernel or user process, there is a page of memory organized as a ring buffer of request slots, called a *post page*. The client process running on the compute processor uses the post page to post requests to the co-processor and receive results back.

The code on the co-processor is a dispatch loop that runs in kernel mode with interrupts disabled. This loop inspects the next active slot in each post page. When it detects a request in one of the post pages, it uses the request number as an index in the post switch table and calls the appropriate send procedure to service the request. After the request is serviced, the co-processor puts the result or error message in some location after the message.

Within the dispatch loop, the co-processor also polls the processor status registers for incoming messages. When a packet arrives, the dispatch loop reads the packet type to index into the packet switch table, and calls the appropriate receive packet procedure.

3.4 Protocol Extensions

To use the co-processor to overlap some SVM protocol operations, we extended the communication co-processor code by adding one more request type and one more packet type to its interface to support the following operations:

Compute Diff. This operation is used in both overlapped protocols. The compute processor specifies the page, page size, and the address of the twin that was previously created with the clean contents of the page. After the co-processor validates the passed parameters, it flips the "done" flag to allow the computation processor to continue, and then starts computing diffs. When the diff computation is complete, the co-processor sends the diff together with the local timestamp to the home node (in the OHLRC case) or just saves the address of the diff in the corresponding write notice record (in the OLRC case).

Apply Diff. This operation is used in the OLRC protocol to receive new diffs. The receiving co-processor transfers the modifications to its local copy, updates the home's flush timestamp for that particular page and processor accordingly, and services pending page requests if they are satisfied by the current version of the page.

Fetch Diffs. This operation is used in the OLRC protocol to collect necessary diffs. The faulting processor submits fetch-diff requests to other nodes for one or multiple diffs. The co-processor of the destination node services the requests when the diff computation is complete.

Fetch Page. The fetch-page operation is used in both overlapped protocols. On a memory access miss, the faulting processor in the OHLRC protocol sends a

page request to the co-processor of the home node with the vector of lock timestamps for that page, whereas in the OLRC protocol the page request is sent to a member in the approximate copyset for that page. The co-processor of the remote node either services this request if the page is ready, or simply puts it in a page request list. The page will be sent out when all the required diffs have arrived and have been applied.

3.5 Synchronization and Garbage Collection

Synchronization handling and related coherence checking for all four prototypes is implemented at user level using NX/2 messages. Each lock has a manager, which is assigned in a round-robin fashion among the processors. The manager keeps track of which processor has most recently requested the lock. All lock acquire requests are sent to the manager unless the node itself holds the lock. The manager forwards the lock request to the processor that last requested the lock. The request message contains the current maximal vector timestamp of the acquiring processor. When the lock arrives, it contains the releaser's knowledge of all time intervals for the requester to update its timestamp vectors.

Barriers are also implemented using a centralized manager algorithm. When a barrier is entered, each node sends the barrier manager the write notices for all intervals that the manager has not seen. The barrier manager collects the intervals from all other nodes, computes the maximal timestamp, and selectively forwards the missing write notices to each node.

For LRC and OLRC, barrier synchronizations trigger garbage collection of protocol data structures when the free memory is below a certain threshold, similar to the approach used in TreadMarks [21]. Garbage collection is quite complex because it needs to collect all "live" diffs, which are distributed on various nodes. All last writers for each individual shared page need to validate the page by requesting all the missing diffs from other nodes. The non-last writers can simply invalidate the page, and modify the copyset for that page. After this phase, the collection algorithm can clean up the heaps and data structures.

For the HLRC and OHLRC protocols, there is no need to perform garbage collection since no diffs or write notices are ever stored beyond a release or barrier.

4 Performance

This section presents performance results for our implementations on five benchmark programs. We begin with a description of the benchmark applications and problem sizes used. We then evaluate the overall performance of the four protocols on these benchmarks, using speedup on 8, 32, and 64 processors as the metric. We conclude by examining the experimental results in more detail, giving execution time breakdowns, communication traffic, and memory requirements.

4.1 Applications

To evaluate the performance of our implementations, we used two kernels (LU and SOR) and three applications (Water-Nsquared, Water-Spatial and Raytrace), four of which (LU, Water-Nsquared, Water-Spatial and Raytrace) are from the Splash-2 suite. Although the names of two of the applications (Water-Nsquared and Water-Spatial) indicate a similar problem, the algorithms and their sharing patterns are different.

Application	Problem Size	Sequential Execution Time (secs)
LU	2048 × 2048	1,028
SOR	1024 × 4096	1,360
Water-Nsquared	4096 molecules	1,130
Water-Spatial	4096 molecules	1,080
Raytrace	Balls4.env(256x256)	956

Table 1: Benchmark applications, problem sizes, and sequential execution times.

LU performs blocked LU factorization of a dense matrix. The matrix is decomposed in contiguous blocks that are distributed to processors in contiguous chunks. Therefore this kernel exhibits coarse-grain sharing and low synchronization to computation frequency, but the computation is inherently unbalanced. The results presented in this section are for a 2048 × 2048 matrix with 32 × 32 blocks.

SOR is a kernel from the suite of programs used in TreadMarks. It corresponds to the red-black successive over-relaxation (SOR) method for solving partial differential equations. The black and red arrays are partitioned into roughly equal size bands of rows, which are distributed among the processors. Communication occurs across the boundary rows between bands and is synchronized with barriers. We ran the kernel with a 1024 × 4096 matrix for 51 iterations starting, as in Treadmarks, with all elements initialized randomly. We chose to run this kernel, in particular, to allow some extreme case comparison between the protocols.

Water-Nsquared simulates a system of water molecules in liquid state, using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of n molecules, and partitioned among processors into contiguous pieces of n/p molecules each. The interesting communication occurs at the end of each step when each processor updates its own n/p molecules and the following $(n/2 - n/p)$ molecules of other processors in the array, using per-partition locks to protect these updates.

Water-Spatial solves the same problem as Water-Nsquared, but uses a spatial directory rather than a brute-force method, making it more suitable for large problems. The 3-d physical space is broken up into cells, and each processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules currently within those cells. A processor reads data from those processors that own cells on the boundary of its partition. Molecules migrate slowly between cells, so the irregularity of the application, although present, has little impact on performance.

Raytrace renders complex scenes in computer graphics using an optimized ray tracing method. The accesses to the scene data, into which rays are shot in this program, are read-only and relatively uninteresting other than the fact that they cause fragmentation. The interesting communication occurs in task stealing using distributed task queues, and in updating pixels in the image plane as part of a task. Both types of access patterns are fine-grained and cause considerable false sharing and fragmentation at the page level. The original Splash-2 application was modified to reorganize the task queues and remove unnecessary synchronization to alleviate the problems observed in [16].

Table 1 shows the problem sizes and their sequential execution times. For all applications we chose relatively large problem sizes, each requiring approximately 20 minutes of sequential execution. Problem sizes were determined by the capabilities of our four prototypes: although the home-based protocols can run larger problems, we chose the largest problems runnable under all protocols and all machine sizes for the sake of comparison.

4.2 Overall Performance: Speedups

Table 2 summarizes the speedups for the LRC, HLRC, OLRC and OHLRC implementations on 8, 32 and 64 nodes. There are two key observations to be made here. First, the home-based LRC protocols (HLRC and OHLRC) clearly outperform their “homeless” counterparts (LRC and OLRC) with one exception (Water-Spatial on 8 node, non-overlapped protocols), in which case the speedups are comparable. These results are consistent with those obtained through simulation in the comparison between LRC and AURC [16]. Second, the performance gap between home and homeless protocols increases dramatically for 32 and 64 processors configurations. This result, which is consistent across all applications, reveals a significant difference in scalability between the two classes of protocols. For instance, the difference in speedups between HLRC and LRC for 64 processors reaches a factor of 1.7 for LU, a factor of 2 for Water Spatial, a factor of 3 for SOR and a factor of almost 6 for Raytrace. For two of these applications (Water Spatial and Raytrace) the speedups under the LRC protocol actually drop when going from 32 to 64 processors. Obviously such insights would have been impossible to guess from the 8-processor runs, where the performance of the home-based and homeless protocols are very close.

The overlapped protocols provide modest improvements over the non-overlapped ones. The range of speedup improvements varies among applications, from as little as 2-3% to as much as 30%.

Summarizing, given the limitations of the Paragon architecture (e.g., large message latency and high interrupt cost, as explained next), all five real Splash-2 applications perform surprisingly well under the home-based protocols, with more than 50% parallel efficiency on 32 nodes, and between 30% and 66% on 64 nodes.

We now turn to a more detailed examination of these results, starting with the determination of the basic operation costs on the Paragon that provide the context in which the results can be better understood.

Application	8 nodes				32 nodes				64 nodes			
	LRC	O	H	OH	LRC	O	H	OH	LRC	O	H	OH
LU	4.4	4.7	5.2	6.0	11.5	13.5	13.9	16.6	15.5	16.8	27.0	29.1
SOR	4.3	4.8	6.4	6.5	13.0	13.6	22.7	23.6	12.3	12.6	35.7	36.9
Water-Nsquared	6.7	7.0	7.0	7.1	11.7	14.0	18.9	21.0			19.2	20.6
Water-Spatial	7.4	7.5	7.4	7.7	14.1	17.1	20	23.5	10.6	11.5	22.6	26.4
Raytrace	6.9	7.1	7.6	7.7	10.6	10.6	26.8	28.0	7.4	7.4	40.6	43.0

Table 2: Speedups on the Intel Paragon with 8, 32, 64 nodes.

4.3 Cost of Basic Operations

Table 3 shows the costs of important basic operations on the Intel Paragon.

Operation	Time in microseconds
Message Latency	50
Page Transfer	92
Receive Interrupt	690
Twin Copy	120
Diff Creation	380-560
Diff Application	0-430
Page Fault	290
Page Invalidation	200
Page Protection	50

Table 3: Timings for basic operations on the Intel Paragon

Using the basic operation costs we can determine the minimum cost (assuming no contention) for a page miss and a lock acquire. In a non-overlapped protocol, like HLRC, a page miss takes at least $290+50+690+92+50=1,172$ microseconds for a full page transfer. In an overlapped protocol, such as OHLRC, the same page miss takes only $290+50+92+50=482$ microseconds. Similarly, a page miss in LRC takes at least $290+50+690+50=1,130$ microseconds without overlapping and 440 microseconds with overlapping for one single-word diff transfer. A remote acquire request, which is intermediated by the home of the lock, costs at least $50+690+50+690+50=1,550$ microseconds. This could be reduced to only 150 microseconds if this service were moved to the co-processor.

4.4 Home Effect

Table 4 shows the frequency of page faults and diff related operations for HLRC and LRC on 8 and 64-nodes. (We do not give results for the overlapped protocols because they are similar to the non-overlapped ones.) There are several "home effects" revealed by this table. First, since the home's copy of the page is eagerly updated, page faults do not occur and diffs are not created at the home. This can lead to substantial protocol savings, particularly if there is a one writer-multiple readers sharing pattern and the writer is chosen as the home. This explains why no diffs are created for LU and SOR.

The other three applications also have reduced page faults and diffs due to the home effect. However, some-

times HLRC creates slightly more diffs than LRC, as shown in the Raytrace application, because of laziness.

Finally, the home-based protocols have fewer diff-applications than the homeless ones because home-based protocols apply diffs eagerly but only once, whereas the homeless protocols may apply diffs multiple times.

4.5 Time Breakdowns

To better understand where time goes, we instrumented our systems to collect the average execution time breakdowns per node. Figure 3 shows the breakdowns, including the computation time, data transfer time, garbage collection time, synchronization time including locks and barriers, and protocol overhead. Protocol overhead includes diff and twin creation, diff application, write-notice handling, and remote request service.

We use Water-Nsquared (Figure 3(c)) as an example to introduce the time breakdown analysis. In both the 8- and 32-node cases, the speedups of HLRC are better than LRC, with the advantage more significant in the 32-node case. The time breakdowns show that the differences are due to the reduction of both lock and barrier synchronization time, of data transfer time, and of protocol overhead.

Synchronization cost dominates the total overhead. To identify the sources of the execution imbalance among processors, we instrumented all applications to collect per processor execution time breakdowns between two consecutive barriers. Figure 4 shows per-processor breakdowns of the execution time for Water-Nsquared between barriers 9 and 10 for both LRC and HLRC protocols on 8 and 64 processors. For 8 processors the imbalance is small and mostly due to computation imbalance. For 64 processors the computation time is evenly distributed among processors; in this case almost all the imbalance is due to lock contention and communication imbalance.

Lock waiting time is significant because page misses occur in critical sections. Therefore variations in the data transfer time are reflected in the lock waiting time as well. For instance, in Water-Nsquared the lock waiting time is larger under LRC than under HLRC because the data transfer time is larger as well. Lock waiting time can cause execution imbalance due to serialization of lock acquisitions when lock contention occurs. For Water-Nsquared, which is a regular application, lock contention occurs when an imbalance in data transfer time occurs. This explains why there is an imbalance in the lock waiting time for LRC but not for HLRC. For irregular applications, like Water Spatial, imbalance in

Application	Number of nodes nodes	Read misses		Diffs Created		Diffs Applied		Lock Acquires	Barriers
		LRC	HLRC	LRC	HLRC	LRC	HLRC		
LU	8	1,050	1,050	5,704	0	0	0	0	64
	64	452	452	1,396	0	0	0	0	128
SOR	8	343	343	25,088	0	171	0	0	98
	64	385	385	3136	0	342	0	0	98
Water Nsquared	8	938	811	1,266	810	4,809	810	652	12
	32	893	859	956	815	8,742	815	556	12
Water Spatial	8	1,361	1,356	466	10	74	10	11	10
	64	918	878	122	66	3,948	66	11	10
Raytrace	8	339	209	551	733	3,591	733	340	1
	64	108	87	83	103	2,368	103	94	1

Table 4: Average number of operations on each node.

the lock waiting time occurs even when data transfer time is balanced.

For regular applications data transfer time imbalance occurs as a result of serialization when multiple data requests arrive at the same processor simultaneously. We call this situation a “hot spot”. Homeless protocols are likely to generate hot spots more frequently than home-based protocols because in the homeless protocols updates usually are collected from the last writer, whereas in the home-based protocols updates are distributed to homes. This situation occurs for instance in Water-Nsquared, which exhibits a multiple-writer multiple-reader sharing pattern with coarse-grained read and writes [16].

The second dominating overhead is data transfer time. The data transfer times of HLRC are smaller than those of LRC for three reasons. First, for regular applications with coarse-grain sharing and migratory data patterns [16], the data traffic to service page misses is higher for the homeless protocol than for the home-based protocol. This is counter-intuitive, since LRC transfers diffs whereas HLRC always transfers full pages. However, migratory data patterns can produce aggregate diff sizes in LRC significantly larger than a page. Second, LRC has to apply the diffs on the faulting page following the happen-before partial order, whereas HLRC does not require any additional processing after fetching. Third, HLRC has fewer page misses because of the home effect.

By offloading some of the protocol overhead from the critical path to the communication processors, the overlapped protocols reduce data transfer time, protocol overhead, and synchronization time. Data transfer time is reduced because the remote fetch requests are handled on the communication processor. Synchronization cost is reduced slightly because overlapping does not change the execution imbalance among the processors. Protocol time is always reduced with overlapping, but its relative contribution to the total execution time is small.

The performance analysis for Water-Nsquared holds for the other applications as well. The 8-processor run of Water-Spatial reveals a case in which HLRC performs slightly worse than LRC. Although Water-Spatial induces a multiple-writer sharing pattern at page level [16], for a small number of steps (two in our case)

most pages are single-writer, so LRC and HLRC are similar in terms of data traffic. However, the message-size limitation of the Paragon, forces HLRC to send more physical messages, thus increasing the data transfer time comparing with LRC.

4.6 Communication Traffic

Table 5 shows the communication traffic of the LRC and HLRC prototypes for all applications. The traffic information gives us several insights that substantiate our time breakdown analysis. Since the overlapped protocols have approximatively the same communication traffic as the non-overlapped ones, we compare only HLRC with LRC.

For each created diff, HLRC always sends one message to update the home. In addition one message is always enough in HLRC to satisfy any page miss. On the other hand, a homeless LRC protocol requires at least one message per page miss. It requires exactly one when there is only one writer per page or when the page contains migratory data (in which case all the diffs are found at the last writer). If there is only one writer per page, as in SOR, the two protocols send about the same number of messages unless LRC is penalized by the garbage collection process, as in LU, in which case it requires additional messages to fetch pages. In a regular migratory data application, like Water-Nsquared, HLRC ends up sending more messages than LRC because of the additional messages sent to update the homes eagerly. In the common case, which corresponds to multiple reader and writer pages, as in Raytrace, LRC sends more messages than HLRC because it requires more than one diff message to satisfy a page miss.

For applications with predominantly single-writer pages, such as LU and Water-Spatial, the amount of update-related traffic in HLRC and LRC is similar because LRC can avoid diff aggregation. In SOR, which also exposes single-writer pages, LRC data traffic is affected by garbage collection. For applications with multiple-writer pages, such as Water-Nsquared or Water-Spatial, on 64 processors the data traffic under HLRC is substantially less than under LRC.

For applications with fine-grain sharing, like Raytrace, the comparison moves towards what we expected

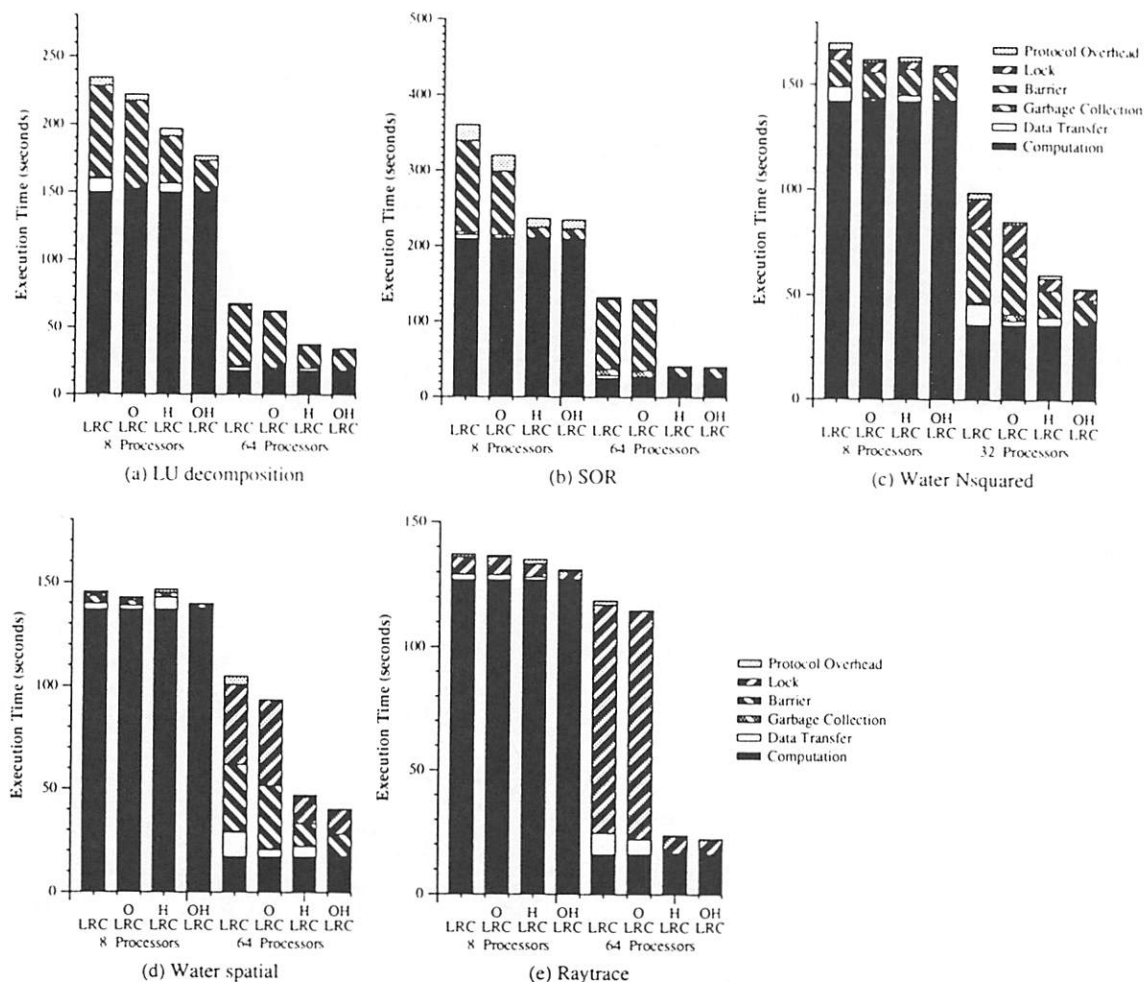


Figure 3: Time breakdowns of LU, SOR, Water-Nsqared, Water-Spatial and Raytrace.

to be the typical comparison pattern between HLRC and LRC: more messages in LRC and more traffic in HLRC. This leads to a latency vs bandwidth tradeoff in the comparison of the two protocols with respect to data transfer time. For instance, systems like the Paragon (as well as ATM networks), which has relatively high message latency and high bandwidth, are likely to benefit more from the home-based protocols.

Finally, for the protocol traffic, the home based LRC approach is consistently cheaper than the homeless LRC protocol. The homeless protocol sends more protocol related data than a home-based one, especially for large number of processors where write notices can become substantially larger due to full vector timestamp inclusion.

In conclusion, home-based protocols scale better than the homeless protocols in terms of protocol traffic.

4.7 Memory Requirements

Protocol memory requirement is an important criterion for scalability. It directly affects application performance in two ways: the limitation on the problem sizes and the frequency of garbage collection of protocol related data.

Table 6 reveals some interesting facts. If a garbage collection is triggered only at barriers (as we actually implemented in our prototypes), then the memory requirement of homeless protocols can be even larger than the application memory (by a factor of 8 in Water-Nsqared). On the other hand, the memory used by the home-based protocols is only a small percentage of the application memory (10% to 25%). As we increase the number of processors, the memory required for protocol data doesn't change much in HLRC, but in LRC it can increase dramatically (Water-Spatial and Raytrace). Although it is possible to reorganize the data structures to reduce the memory consumption, we do not expect that the overall picture would change dramatically.

Out of the various protocol data structures, the major memory consumers are the diffs and the write notices. In HLRC, diffs are discarded almost immediately after they are created or applied, while in LRC they have to be kept for an indefinite amount of time (until the garbage collection time in most cases). Write notices can consume a large amount of memory. For instance, the LRC protocol consumes about 5 Mbytes of memory on each of the 32 processors to store write notices in the Water-Nsqared application. Since the write-notice data structure includes the full vector timestamps in the

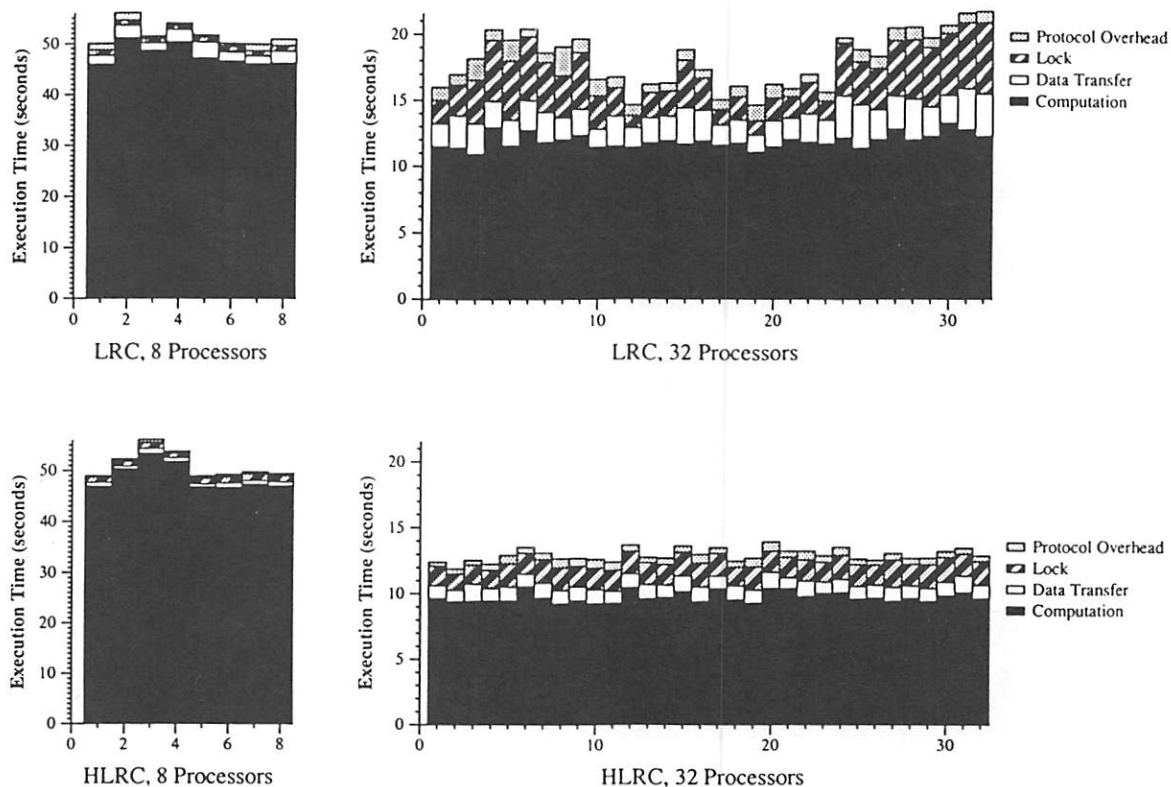


Figure 4: Time breakdowns of Water-Nsquared between barriers 9 and 10.

homeless protocols, the storage requirement increases proportionally with the number of processors.

In short, home-based protocols scale much better than the homeless ones with respect to memory requirements.

4.8 Discussion

An interesting question is whether home-based protocols are always better than homeless ones. The performance evaluation we have conducted on the Paragon provides limited answers towards a complete understanding of the tradeoffs between home-based and homeless LRC protocols. The Paragon architecture is characterized by message latency, page fault, and interrupt times that are relatively large compared with memory and network bandwidth (Table 3). As a consequence, a roundtrip communication for either a page or lock transfer is at best on the order of a millisecond. Current network technologies [6, 13, 7], as well as aggressive software for fast interrupts, exceptions [30] and virtual memory mapped communication [10, 11] have brought such latencies down significantly to the neighborhood of a couple of microseconds. An interesting question is to what extent our results are specific to the Paragon architecture and how they would be affected by different architectural parameters. Fast interrupts and low latency messages make the roundtrips shorter and since LRC has usually more messages than HLRC it is likely that the homeless protocols will benefit more from these architectural improvements. Therefore, the performance gap between the home-based and the homeless protocols

would probably be smaller on such architectures.

Our performance evaluation shows that home-based protocols perform much better than the homeless ones for three regular applications with coarse-grain sharing and for two irregular applications with small communication to computation ratios. The traffic ratio for the two classes of protocols will probably be different if fine-grain sharing dominates. To help answer this question we ran SOR with all elements 0, except at the edges. Under this initialization, the interior elements of the matrix do not change for the first many iterations. Consequently, these pages have no updates during those intervals, and so do not produce diffs. Even when diffs are produced later in the computation, there is only a single writer per page, a single diff of produced per interval, and the diff size increases gradually. Consequently, the conditions of this experiment favor LRC over HLRC, which must transfer full pages. Despite these factors, the experimental results show that HLRC is still 10% better than LRC. This experiment suggests that HLRC is likely to have robust performance behavior for a large number of applications.

4.9 Limitations

In addition to the Paragon communication parameters there are several specific limitations in our experiments. The virtual memory page size used in the OSF-1 operating system is 8 Kbytes, although the hardware allows 4 Kbyte page size. We have not been able to modify the virtual memory system to change the page size to conduct more experiments. Asynchronous

Application	Number of nodes	Update traffic				Protocol traffic			
		Number of messages		Message traffic (Mbytes)		Number of messages		Message traffic (Mbytes)	
		LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC
LU	8	1,050	1,050	8.6	8.6	1,178	1,178	0.28	0.16
	64	452	452	3.7	3.7	708	708	1.7	0.6
SOR	8	343	343	4.2	2.8	538	538	0.95	0.76
	64	385	385	6.0	3.2	581	581	2.6	0.9
Water-Nsquared	8	1,102	1,621	16.1	6.8	2,784	2,531	1.8	0.8
	32	1,421	1,674	11.5	7.1	3,093	2,493	6.7	1.2
Water-Spatial	8	1,360	1,366	7.3	7.3	1,411	1,403	0.06	0.03
	64	2,581	944	16.4	4.7	2,628	925	0.42	0.08
Raytrace	8	1,069	942	1.46	1.43	1,446	586	0.44	0.22
	64	2,368	190	0.10	0.61	403	313	0.52	0.20

Table 5: Average communication traffic on each node.

Application (memory used)	Number of nodes	Protocol data structure memory used (Mbytes)									
		Twins		Diffs		Page table		Write notices		Total	
		LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC	LRC	HLRC
LU (32 Mbytes)	8	4.2	0	4.4	0	1.3	2.6	0.1	0.1	10.0	2.8
	64	2.5	0	2.7	0	1.3	2.6	1.3	0.2	7.8	2.8
SOR (33 Mbytes)	8	4.2	0	8.4	0	1.3	2.6	0.13	0.008	14.3	2.6
	64	2.3	0	5.2	0	1.3	2.6	0.9	0.009	9.8	2.6
Water-Nsquared (2.5 Mbytes)	8	7.3	0.3	10.8	0	0.1	0.2	1.8	0.04	20.0	0.54
	32	3.9	0.1	4.2	0	0.1	0.2	5.4	0.15	13.6	0.45
Water-Spatial (4.8 Mbytes)	8	2.8	0.5	6.4	0	0.2	0.4	0.12	0.003	9.5	0.9
	64	0.4	0.1	12.0	0	0.2	0.4	0.4	0.01	13.0	0.5
Raytrace (10.2 Mbytes)	8	0.4	0.4	0.6	0	0.3	0.6	0.8	0.1	2.1	1.1
	64	0.7	0.1	13	0	0.3	0.7	1.3	0.1	15.2	0.9

Table 6: Average memory requirements on each node.

receives are not interruptible. The Paragon NX/2 message layer cannot support message packets larger than the page size. This limitation can affect both HLRC and LRC. HLRC has always to send two messages in reply of a page fetch request: one with the page and the other with the timestamps of the page. In LRC when the aggregate diff size exceeds the page size, the number of actual messages sent is larger than the number of logical messages.

Several possible optimizations are still unexplored. Synchronization operations have been implemented in a straightforward way using NX/2 messages and centralized algorithms. Overlapping using the communication processor was not employed for coherence and synchronization control. Finally, we have reported the results for only five applications and for a single problem size.

5 Related Work

Since shared virtual memory was first proposed ten years ago [26], a lot of work has been done on it. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence [12]. The model was used to implement shared virtual memory and reduce false sharing by allowing multiple writers [8]. Lazy Release Consistency (LRC) [22, 9, 1] further relaxed

the RC protocol to reduce protocol overhead. TreadMarks [21] was the first SVM implementation using the LRC protocol on a network of stock computers. That implementation has achieved respectable performance on small-scale machines.

The recently proposed Automatic Update Release Consistency protocol (AURC) [15] is an LRC protocol that takes advantage of the automatic update mechanism in virtual memory-mapped communication. The idea of using a home-based approach to build an all-software protocol similar to AURC was proposed in [17]. Our home-based LRC protocols are based on the AURC protocol, but the updates are detected in software using diffs, as in the standard LRC. A degree of overlapping similar to the one the automatic update mechanism provides is achieved in our Overlapped Home-based LRC (OHLRC) protocol, where the communication co-processor is used to perform, transfer, and apply the diffs.

In a recent work [23], Keleher has shown that a simple single-writer LRC protocol perform almost as well as a more complicated multiple-writer LRC. His protocol totally eliminates diff-ing at the expense of a higher bandwidth requirement for full page transfers. Our home-based protocols support multiple writers using diffs but replace most of the diff traffic with full page traffic. The home-based protocols reduce to a single-

writer protocol for applications that exhibit one-writer multiple-readers sharing patterns, like SOR or LU.

Other relaxed consistency models include Entry Consistency [3] and Scope Consistency [17]. Both models take advantage of the association of data with locks, either explicitly (Entry Consistency) or implicitly (Scope Consistency), to reduce the protocol overhead. Both Orca [2] and CRL [19] are designed to implement distributed shared memory by maintaining coherence at object level instead of page level. These methods require specialized APIs, unlike the prototype systems presented in this paper. Our systems allow programs written for a release-consistent, shared-memory multiprocessor to run without modification.

Several multicomputers use a dedicated co-processor for communication on each node. Examples include the Intel Paragon [27] and the Meiko CS-2 [14]. The Typhoon [29] system uses a special hardware board to detect access faults at fine granularity and implements distributed shared memory on a network of HyperSparc workstations. It uses one of the two CPUs in the dual-processor workstation as a protocol processor. In the Flash multiprocessor [24], each node contains a programmable processor called MAGIC that performs protocol operations and handles all communications within the node and among all nodes. Neither system uses LRC-based relaxed consistency models.

Bianchini et al. [4] proposed a dedicated protocol controller to offload some of the communication and coherence overheads from the computation processor. Using simulations they show that such a protocol processor can double the performance of TreadMarks on a 16-node configuration and that diff prefetching is not always beneficial. The protocol they evaluate is similar to our overlapped homeless LRC protocol (OLRC).

A recent study [20] investigated how to build an SVM system on a network of SMPs. They studied the tradeoffs of using a dedicated processor or the spare cycles of a compute processor to execute coherence protocol. The study is limited to simulations.

6 Conclusions

This paper proposes two new home-based protocols based on Lazy Release Consistency (LRC): Home-based HLRC (HLRC) and Overlapped Home-based LRC (OHLRC). Our experiments with five applications on a 64-node Intel Paragon multicomputer show that the home-based protocols perform and scale substantially better than their homeless counterparts. To our knowledge this is the first performance study of a page-based software shared memory system on such a large configuration. We have also found that protocol overlapping using the communication processor provides only modest performance improvement.

The HLRC protocol outperforms the standard LRC protocol for several reasons:

- Synchronization cost is higher for the homeless protocols. This is because the waiting time depends to a large extent on the data transfer time, which is both larger and possibly imbalanced in the

homeless protocols. For all applications, the synchronization cost is the main component limiting performance for large configurations.

- Data transfer time is also larger for the homeless LRC protocols. Data traffic is usually larger in LRC because the aggregate diff size fetched on a page miss in LRC can exceed the size of a page, the fixed transfer size in HLRC. Also, the number of messages is usually larger in LRC. Finally, hot spots are likely to occur more frequently for a homeless protocol.
- A home-based protocol is simpler than a homeless one. As a consequence, HLRC produces less protocol overhead, generates less protocol traffic, and requires substantially less memory than LRC. Diff garbage collection is not required in HLRC because diffs have a very short lifetime in a home-based protocol.

An interesting question is whether home-based protocols are always better than homeless ones. While our study provides only limited answers, it suggests that home-based protocols are likely to perform robustly for a large number of applications.

Acknowledgments

We thank our "shepherd" John Zahorjan for his generous and thoughtful help that substantially improved this paper. This work benefitted greatly from discussions the authors had with Jaswinder Pal Singh who suggested to us to pursue the comparison with homeless protocols. We are grateful to Doug Clark who encouraged us to pursue this study as well as to Czarek Dubnicki and Yuqun Chen for their aid in understanding the Paragon kernel. Pete Keleher and Sandhya Dwarkadas helped us to understand the garbage collection in TreadMarks. We also thank the anonymous reviewers for their insightful comments. Hongzhang Shan provided us with an improved version of Raytrace. We thank Matthias Blumrich for his careful reading of the draft. We want to thank Paul Messina, Heidi Lorenz-Wirzba and Sharon Brunett from the Caltech CCSC Division for providing us with the precious Paragon cycles without which this paper would not have existed.

This project is sponsored in part by ARPA under contract under grant N00014-95-1-1144 and DABT63-94-C-0049, by NSF under grant MIP-9420653, and by Intel Corporation.

References

- [1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.
- [2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. A Distributed Implementation of the Shared Data-Object Model.

In *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, October 1989.

- [3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.
- [4] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.
- [6] Matthias Blumrich, Cezary Dubnick, Edward Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.
- [7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
- [8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.
- [9] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.
- [10] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.
- [11] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.
- [12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.
- [13] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.
- [14] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proceedings of Hot Interconnects*, August 1993.
- [15] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.
- [16] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.
- [17] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.
- [18] i860TM XP Microprocessor. Programmer's Reference Manual, 1991.
- [19] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 213–228, December 1995.
- [20] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.
- [21] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.
- [22] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.
- [23] P.J. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the IEEE COMPCON '96 Conference*, February 1996.
- [24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford Flash Multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 302–313, April 1994.
- [25] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, May 1992.
- [26] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.
- [27] R. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994.
- [28] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.
- [29] James R. Larus Steven K. Reinhardt and David A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.
- [30] Ch. A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.
- [31] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

Invited Talk: Active Networks

David Tennenhouse, *Telemedia, Networks and Systems Group,*
Laboratory for Computer Science, MIT

Abstract:

Active networks allow individual users, or groups of users, to inject customized programs into the nodes of the network. This approach to network architecture will enable a massive increase in the sophistication and customization of the computation that is performed within the network. In this talk I will discuss the overall motivation for our work, the approach we have adopted, and a prototype implementation that leverages the existing IP options mechanism.

Our work is motivated by both technology push and user pull. The “pull” comes from the ad hoc collection of fire walls, web proxies, multicast routers, mobile proxies, video gateways, etc., that perform user-driven computation at nodes within the network. We are developing the architectural support and common programming platforms to support the diversity and dynamic deployment requirements of these inter-posed services.

The “push” is the emergence of active technologies, compiled and interpreted, supporting the encapsulation, transfer, inter-position, and safe and efficient execution of program fragments. Today, these technologies are applied above the end-to-end network layer, e.g., to allow clients and servers to exchange fragments. Our innovation is to leverage and extend these technologies for use within the network in ways that will fundamentally change today’s model of what is “in” the network.

Architecturally, we are bumping up the level of abstraction at which inter-operability is achieved: today’s routers support the agreed syntax and semantics of IP, e.g., they all perform “equivalent” computations. In contrast, active nodes perform many different computations but must all support an equivalent and inter-operable computational model.

David L. Tennenhouse is a Principal Research Scientist at MIT's Laboratory for Computer Science and Sloan School of Management. He is leader of the Telemedia, Networks and Systems Group, which is addressing 'systems' issues arising at the boundaries between telecommunications and distributed computing. David has a B.A.Sc. and an M.A.Sc in Electrical Engineering from the University of Toronto and a Ph.D. from the University of Cambridge.

CPU Inheritance Scheduling

Bryan Ford Sai Susarla

*Department of Computer Science
University of Utah
Salt Lake City, UT 84112*

flux@cs.utah.edu

<http://www.cs.utah.edu/projects/flux/>

Abstract

Traditional processor scheduling mechanisms in operating systems are fairly rigid, often supporting only one fixed scheduling policy, or, at most, a few “scheduling classes” whose implementations are closely tied together in the OS kernel. This paper presents *CPU inheritance scheduling*, a novel processor scheduling framework in which arbitrary threads can act as schedulers for other threads. Widely different scheduling policies can be implemented under the framework, and many different policies can coexist in a single system, providing much greater scheduling flexibility. Modular, hierarchical control can be provided over the processor utilization of arbitrary administrative domains, such as processes, jobs, users, and groups, and the CPU resources consumed can be accounted for and attributed accurately. Applications, as well as the OS, can implement customized local scheduling policies; the framework ensures that all the different policies work together logically and predictably. As a side effect, the framework also cleanly addresses priority inversion by providing a generalized form of priority inheritance that automatically works within and among diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity[29] and scheduler activations[3]. We show that this flexibility can be provided with acceptable overhead in typical environments, depending on factors such as context switch speed and frequency.

1 Introduction

Traditional operating systems control the sharing of the machine’s CPU resources among threads using a fixed scheduling scheme, typically based on priorities. Sometimes a few variants on the basic policy are provided,

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army, under contract number DABT63-94-C-0058. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies of the U.S. Government.

such as support for fixed-priority threads[17], or several “scheduling classes” to which threads with different purposes can be assigned (e.g., real-time, interactive, or background)[25]. However, even these variants are generally hard-coded into the system implementation and cannot easily be adapted to the needs of individual applications.

In this paper we develop a novel processor scheduling framework based on a generalized notion of priority inheritance. In this framework, known as *CPU inheritance scheduling*, arbitrary threads can act as schedulers for other threads by temporarily *donating* their CPU time to selected threads while waiting on events of interest such as clock/timer interrupts. The receiving threads can further donate their CPU time to other threads, and so on, forming a logical hierarchy of schedulers as illustrated in Figure 1. A scheduler thread can be notified when the thread to which it donated its CPU time no longer needs it (e.g., because the target thread has blocked), so that the CPU can be reassigned to another target. The basic thread dispatching mechanism necessary to implement this framework does not have any notion of thread priority, CPU usage, or clocks and timers; all of these functions, when needed, are implemented by threads acting as schedulers.

Under this framework, arbitrary scheduling policies can be implemented by ordinary threads cooperating with each other through well-defined interfaces that may cross protection boundaries. For example, a fixed-priority multiprocessor scheduling policy can be implemented by maintaining, among a group of scheduler threads (one for each available CPU), a prioritized queue of “client” threads to be scheduled; each scheduler thread successively picks a thread to run and donates its CPU time to the selected target thread while waiting for an interesting event such as quantum expiration (e.g., a clock interrupt). See Figure 2 for an illustration. If the selected thread blocks, its scheduler thread is notified and the CPU is reassigned. On the other hand, if a different event causes the scheduler thread to wake up, the running thread is preempted and the CPU is given back to the scheduler immediately. Other scheduling policies, such as timesharing[23], fixed-priority[12, 17],

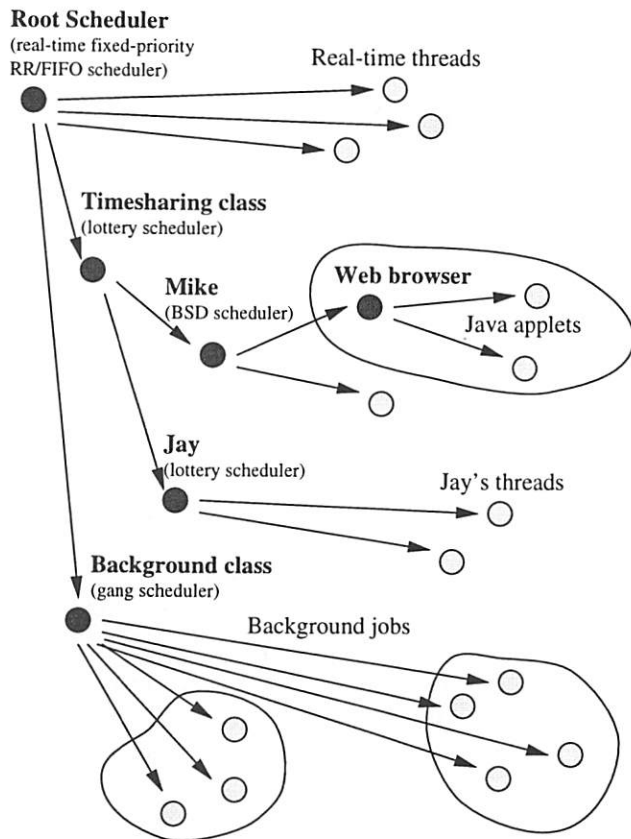


Figure 1: Example scheduling hierarchy. The dark circles represent threads acting as schedulers, while the light circles represent “ordinary” threads.

rate monotonic[22], fair share[9, 16, 19], and lottery/stride scheduling[30, 31, 32], can be implemented in the same way.

This scheduling framework has the following features:

- It supports multiple arbitrary scheduling policies on the same or different processors.
- Since scheduler threads may run either in the OS kernel or in user mode, applications can easily extend or replace the scheduling policies built into the OS.
- It provides hierarchical control over the processor resource usage of different logical or administrative domains in a system, such as users, groups, individual processes, and threads within a process.
- CPU usage accounting can be provided to various degrees of accuracy depending on the resources one is willing to invest.
- Priority inversion is addressed naturally in the presence of resource contention, without the need for explicit priority inheritance/ceiling protocols.
- CPU use is attributed properly even in the presence of priority inheritance.

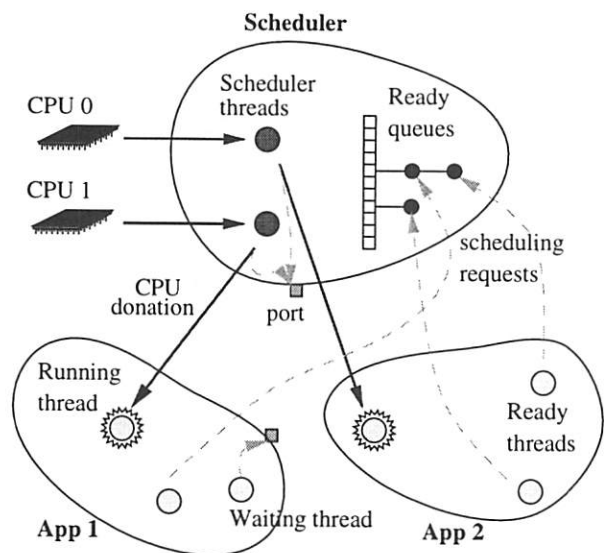


Figure 2: Example fixed-priority scheduler

- The framework naturally extends to multiprocessors.
- Processor affinity scheduling is supported.
- Scheduler activations can be implemented easily.

Based on a prototype implementation in a user-level threads package, we demonstrate that this framework behaves identically to traditional multi-class schedulers in equivalent configurations, and that it provides modular resource control, load insulation, and priority inversion protection automatically across multiple scheduling policies. The additional scheduling overhead caused by this framework is negligible in the test environment, and we show that it should also be acceptable in many kernel environments as well despite greater context switch costs.

The rest of this paper is organized as follows. Section 2 describes motivation and related work, Section 3 presents the CPU inheritance scheduling algorithm in detail, and Section 4 shows how traditional scheduling algorithms can be implemented on this framework. Section 5 demonstrates the behavior and performance properties of our framework through experimental results, and Section 6 concludes with a short reflective summary.

2 Motivation

Traditional operating systems divide a machine’s CPU resources among threads using a fixed scheduling scheme, typically based on priorities. However, the requirements imposed on an operating system’s scheduler often vary from application to application. For example, for interactive applications, response time is usually the most critical factor—i.e., how quickly the program responds to the user’s commands. For batch jobs, throughput is of primary

importance but latency is a minor issue. Hard real-time applications require meeting application-specific deadlines, while for soft real-time applications, missing a deadline is unfortunate but not catastrophic. There is no single scheduling scheme that works well for all applications.

Over the years, the importance of providing a variety of scheduling policies on a single machine has waxed and waned, following hardware and application trends. In the early years of computing, use of the entire machine was limited to a single user thread; this evolved into multiprogrammed machines in which a single scheduling policy managed batch jobs effectively. The advent of timesharing on machines still used for batch jobs caused a need for two scheduling policies. As timesharing gradually gave way to single-user workstations and PCs, a single scheduling policy was again adequate for a while.

Supporting multiple scheduling policies is again becoming important, not because of multiple *users* on a system, but because of the variety of concurrent *uses* to which modern systems are put. Multimedia drives the need for soft real-time scheduling policies on general purpose workstations. Untrusted executable content (e.g., Java applets) requires secure control of resource usage while also providing soft real-time guarantees. The hard real-time domain is also making inroads onto general purpose machines as processors and instruments supporting embedded applications become networked, and some customers (e.g., the military) need the ability to shift processing power dynamically to the problem of the moment. All of these additional policies must still work alongside traditional interactive and batch scheduling: though multimedia and video conferencing may be the current rage, this does not mean that users no longer care about getting good interactive response from the word processors, spreadsheets, and other applications that they still rely on daily. Therefore, as the diversity of applications increases, operating systems need to support multiple coexisting processor scheduling policies, in order to meet individual applications' needs as well as to utilize the system's processor resources more efficiently.

2.1 Related Work

One simple approach to providing real-time support in systems with traditional timesharing schedulers, which has been adopted by many commonly-used systems such as Unix, Mach[1, 4], and Windows NT[25], and has even become part of the POSIX standard[17], is support for fixed-priority threads. Although these systems generally still use conventional priority-based timesharing schedulers, they allow real-time applications to disable the normal dynamic priority adjustments on threads specifically designated as "real-time threads," so that those threads always run at a programmer-defined priority. By carefully assigning priorities to the real-time threads in the system and ensuring that all non-real-time threads execute at lower priority, it

is possible to obtain the correct behavior for some applications. However, this approach has serious, well-known limitations; in many cases, entirely different non-priority-based scheduling policies are needed[18].

Even in normal interactive and batch-mode computing, traditional priority-based scheduling algorithms are showing their age. For example, these algorithms do not provide a clean way to encapsulate a set of processes/threads as a single unit to isolate and control their processor usage relative to the rest of the system. This shortcoming opens the system to various denial-of-service attacks, the most well-known being the creation of a large number of threads which overwhelm processor resources and crowd out other activity. These vulnerabilities generally don't cause serious problems for machines only used by one person, or when the users of the system fall into one administrative domain and can "complain to the boss" if someone is abusing the system. However, as distributed and mobile computing becomes more prevalent and administrative boundaries become increasingly blurred, this form of system security is becoming more important. This is especially true when completely unknown, untrusted code is downloaded and run in a "secure" environment such as that provided by Java[13] or OmniWare[2]. Schedulers have been designed that address this problem by providing flexible hierarchical control over CPU usage at different administrative boundaries[5, 14, 15, 30, 31, 32]. However, it is not yet clear how these algorithms will address other needs, such as those of hard real-time applications; certainly it seems unlikely that a single "holy grail" of scheduling policies will be found.

With the growing diversity of application needs and scheduling policies, it is increasingly desirable for an operating system to support multiple independent policies. On multiprocessor systems, one simple but limited way of doing this is to run a different scheduling policy on each processor. A more general approach is to allow multiple "scheduling classes" to run on a single processor, with a specific scheduling policy associated with each class. The classes have a strictly ordered priority relationship to each other, so the highest-priority class gets all the CPU time it wants, the next class gets any CPU time left unused by the first class, etc. Although this approach shows promise, one drawback is that since the schedulers for the different classes generally don't communicate or cooperate with each other closely, only the highest-priority scheduling class on a given processor can make any assumptions about how much CPU time it will have to dispense to the threads under its control.

Additionally, most existing multi-policy scheduling mechanisms still require every scheduling policy to be implemented in the kernel and to be closely tied with other kernel mechanisms such as threads, context switching, clocks, and timers. The only existing system we know of that allows different scheduling policies to be imple-

mented in separate, unprivileged protection domains is the Aegis Exokernel[8]. However, the Aegis scheduling mechanism was not described at length and does not address important issues such as timing, CPU usage accounting, and multiprocessor scheduling. Both Aegis's scheduling mechanism and our framework are based on the use of a "directed yield" primitive which allows application-level threads to schedule each other; this core concept originally comes from coroutines[6], in which directed yield is the *only* way thread switching is done. We believe our scheduling framework could be implemented in an Exokernel environment through the use of suitable kernel primitives, application-level support code, and standardized inter-process scheduling protocols.

Finally, most existing systems still suffer from various priority inversion problems. Priority inversion occurs when a high-priority thread requesting a service has to wait arbitrarily long for a low-priority thread to finish being serviced. This problem can be addressed in priority-based scheduling algorithms by supporting priority inheritance[7, 26], wherein the thread holding up the service inherits the priority of the highest priority thread waiting for service. In some cases this approach can be adapted to other scheduling policies, such as with ticket transfer in lottery scheduling[31]. However, the problem of resolving priority inversion between threads of different scheduling classes using policies with different and incomparable notions of "priority" has not been addressed so far.

3 CPU Inheritance Scheduling

In our scheduling model, as in traditional systems, a *thread* is a virtual CPU whose purpose is to execute instructions. A thread may or may not have a real CPU assigned to it at any given instant; a running thread may be preempted and its CPU reassigned to another thread at any time. (For the purposes of this framework, it is not important whether these threads are kernel-managed or user-managed threads, or whether they run in supervisor or user mode.)

In traditional systems, threads are generally scheduled by some lower-level entity, such as a scheduler in the OS kernel or a user-level threads package. In contrast, the basic idea of CPU inheritance scheduling is that threads are scheduled by *other threads*. Any thread that has a real CPU available to it at a given instant can *donate* its CPU temporarily to another specific thread instead of using the CPU itself. This operation is similar to priority inheritance in conventional systems, except that it is done explicitly by the donating thread, and no notion of "priority" is involved, only a direct transfer of the CPU from one thread to another; hence the name "CPU inheritance."

A *scheduler thread* is a thread that spends most of its time donating its CPU resources to *client threads*; the client threads thus *inherit* some portion of the scheduler thread's CPU resources, and treat that portion as *their* virtual CPU.

Client threads can in turn act as scheduler threads, distributing their CPU time among their own clients, and so on, forming a scheduling hierarchy.

The only threads in the system that inherently have actual CPU time available to them are the set of *root scheduler threads*; other threads can only run if CPU time is donated to them. There is one root scheduler thread for each real CPU in the system; each CPU is permanently dedicated to supplying CPU time to its associated root scheduler thread. The actions of the root scheduler thread on a given CPU determine the base-level scheduling policy for that CPU.

3.1 The Dispatcher

Even though all high-level scheduling decisions are performed by threads, a small low-level mechanism is still needed to implement primitive thread management functions. We call this low-level mechanism the *dispatcher* to distinguish it clearly from high-level schedulers.

The role of the dispatcher is to implement thread blocking, unblocking, and CPU donation. The dispatcher fields events and directs them to threads waiting on those events, without actually making any real scheduling decisions. Events can be synchronous, such as an explicit wake-up of a sleeping thread by a running thread, or asynchronous, such as an external interrupt (e.g., I/O or timer). The dispatcher itself is not a thread; it merely runs in the context of whatever thread owns the CPU at the time an event of interest occurs.

The dispatcher inherently contains no notion of thread priorities, CPU usage, or even clocks and timers. In a kernel supporting CPU inheritance scheduling, the dispatcher is the only scheduling component that *must* be in the kernel; all other scheduling code could in theory run in user-mode threads outside of the kernel (although this "purist" approach may be impractical for performance reasons).

3.2 Requesting CPU Time

Because no thread (except a root scheduler thread) can ever run unless some other thread donates CPU time to it, a newly-created or newly-woken thread must request CPU time from some scheduler before it can run. Each thread has an associated scheduler which has primary responsibility for providing CPU time to the thread. When the thread becomes ready, the dispatcher notifies the thread's scheduler that the thread needs CPU time. The exact form such a notification takes is not important; in our implementation, notifications are simply IPC messages sent by the dispatcher to Mach-like message ports.

When a thread wakes up, the notification it produces may in turn wake up a scheduler thread waiting to receive such messages on its port. Waking up that scheduler thread will

cause another notification to be sent to *its* scheduler, which may wake up still another thread, and so on. Thus, waking up an arbitrary thread can cause a chain of wakeups to propagate back through the scheduler hierarchy. Eventually, this propagation may wake up a scheduler thread that is currently being supplied with CPU time but is donating it to some other thread. In that case, the thread currently running on that CPU is preempted and control is given back to the woken scheduler thread immediately; the scheduler can then make a decision to re-run the preempted client, switch to the newly-woken client, or even run some other thread. Alternatively, the propagation of wake-up events may terminate at some point, for example because a notified scheduler is already awake (not waiting for messages) but has been preempted. In that case, the dispatcher knows that the wake-up event is irrelevant for scheduling purposes at the moment, so the currently running thread is resumed immediately.

3.3 Relinquishing the CPU

At any time, a running thread may block to wait for one or more events to occur, such as I/O completion or timer expiration. When a thread blocks, the dispatcher returns control of the CPU to the scheduler thread that provided it to the running thread. That scheduler may then choose another thread to run, or it may relinquish the CPU to *its* scheduler, and so on up the line until some scheduler finds work to do.

3.4 Voluntary Donation

Instead of simply blocking, a running thread can voluntarily donate its CPU to another thread while waiting on an event of interest; this is done in situations where priority inheritance would traditionally be used. For example, when a thread attempts to obtain a lock that is already held, it may donate its CPU time to the thread holding the lock; similarly, when a thread makes an RPC to a server thread, the client thread may donate its CPU time to the server for the duration of the request. When the event of interest occurs, the donation ends and the CPU is given back to the original thread. In our implementation of this framework, the basic synchronization and IPC primitives automatically invoke the dispatcher to perform voluntary donation appropriately when the thread blocks; however, voluntary donation could also be done optionally or through explicit dispatcher calls.

It is possible for a single thread to inherit CPU time from more than one source at a given time: for example, a thread holding a lock may inherit CPU time from several threads waiting on that lock in addition to its own scheduler. In this case, the effect is that the thread has the opportunity to run at any time *any* of its donor threads would have been able to run. A thread only “uses” one CPU source at a time; however, if its current CPU source runs

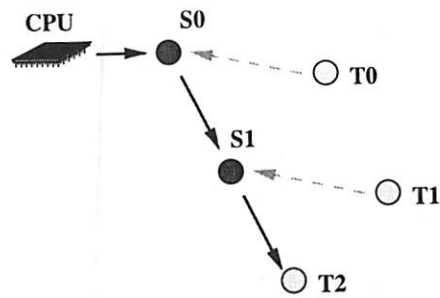


Figure 3: CPU donation chain

out (e.g., due to quantum expiration), the dispatcher will automatically send scheduling request notifications on behalf of all the threads depending on (donating to) to the preempted thread, effectively switching the thread automatically to another available CPU source. One potential worry is that a thread consuming CPU time from many sources will cause an “avalanche effect” every time it is preempted or woken as the dispatcher fires off several scheduling requests, each of which may cause more scheduling requests as intermediate-level schedulers are woken up. We believe that in practice it should be uncommon for a thread to inherit from more than one or two other threads at once, so this should not be a major problem; however, we have not yet examined this issue in detail.

3.5 The schedule operation

The call a scheduler thread makes to donate CPU time to a client thread is simply a special form of voluntary CPU donation, in which the thread to donate to and the event to wait for can be specified explicitly. In our implementation, the `schedule` operation takes as parameters a thread to donate to, a port on which to wait for messages from other client threads, and a *wakeup sensitivity* parameter indicating in what situations the scheduler should be woken. The operation donates the CPU to the specified target thread and puts the scheduler thread to sleep on the specified port; if a message arrives on that port, such as a notification that another client thread has been woken or a message indicating that a timer has expired, then the `schedule` operation terminates and control is returned to the scheduler thread.

In addition, the `schedule` operation may be interrupted before a message arrives, depending on the behavior of the thread to which the CPU is being donated and the value of the wakeup sensitivity parameter. The wakeup sensitivity level acts as a hint to the dispatcher allowing it to avoid waking up the scheduler thread except when necessary; it is only an optimization and is not in theory required for the system to work. Our system supports the following three sensitivity levels:

- `WAKEUP_ON_BLOCK`: If the target of the `schedule` operation blocks without further donating the CPU,

then the schedule operation terminates and control is returned to the scheduler immediately. For example, in Figure 3, if scheduler thread S_1 has donated the CPU to thread T_2 using this wakeup sensitivity setting, but T_2 blocks and can no longer use the CPU, then S_1 will receive control again. WAKEUP_ON_BLOCK is the “most sensitive” setting, and is typically used when the scheduler has other client threads waiting to run.

- **WAKEUP_ON_SWITCH**: If the client thread using the CPU (T_2) blocks, control is *not* immediately returned to its scheduler (S_1): the dispatcher behaves instead as if S_1 itself blocked, and passes control on back to *its* scheduler, S_0 . If T_2 is subsequently woken up, then when S_0 again provides the CPU to S_1 , the dispatcher passes control directly back to T_2 without actually running S_1 . However, if a *different* client of S_1 , such as T_1 , wakes up and sends a notification to S_1 's message port, then S_1 's schedule operation *will* be interrupted. This sensitivity level is typically used when a scheduler has only one thread to run at the moment and doesn't care when that thread blocks or unblocks, but it still wants to switch between different client threads manually: for example, the scheduler may need to start and stop timers when switching between client threads.
- **WAKEUP_ON_CONFLICT**: The scheduler is only awakened if a *second* client thread wakes up while the scheduler is already donating CPU to a client (e.g., if T_1 wakes up while T_2 is running). If T_2 blocks, the scheduler blocks too; then, if *any* single client of scheduler S_1 is subsequently woken, such as T_1 , the dispatcher passes control directly to the woken client thread without waking up the scheduler. At this weakest sensitivity level, the dispatcher is allowed to switch among client threads freely; the scheduler only acts as a “conflict resolver,” making decisions when two client threads are runnable at the same time.

4 Implementing High-level Schedulers

This section describes how the basic CPU inheritance mechanism can be used to implement high-level scheduling policies and related features such as CPU usage accounting, processor affinity, and scheduler activations.

4.1 Single-CPU Schedulers

Figure 4 shows a simplified code fragment from a basic non-prioritized FIFO scheduler in our system. The scheduler keeps a queue of client threads waiting for CPU time, and successively runs each one using the `schedule` operation while waiting for messages to arrive on its port (e.g.,

```
void fifo_scheduling_loop()
{
    cur_thread = NULL;
    more_msgs = 1;
    for (;;) {
        if (more_msgs) {
            more_msgs = msg_rcv(&fifo_pset, &msg);
        } else {

            /* Select the thread to run next. */
            if ((cur_thread == 0) &&
                !queue_empty(&fifo_runq))
                cur_thread = q_remove(&fifo_runq);

            /* Select wakeup sensitivity level. */
            cond = q_is_empty(&fifo_runq) ?
                WAKEUP_ON_CONFLICT : WAKEUP_ON_BLOCK;

            /* Schedule and wait for messages. */
            if (cur_thread != NULL)
                more_msgs = schedule(fifo_pset, &msg,
                                    cur_thread, cond);
            else
                more_msgs = msg_rcv(fifo_pset, &msg);
        }

        /* Process the received message. */
        switch (msg.request_code) {
            case MSG_SCHED_REQUEST:
                /* A client thread wants to run. */
                q_enter(&fifo_runq, msg.thread_id);
                break;
            case MSG_SCHED_BLOCKED:
                /* Last thread gave up the CPU. */
                cur_thread = 0;
                break;
        }
    }
}
```

Figure 4: Example single-processor FIFO scheduler.

notifications from newly-woken client threads). When there are no client threads waiting to be run, the scheduler uses the ordinary non-donating `msg_rcv` operation instead of the `schedule` operation in order to relinquish the CPU while waiting for messages. If there is only one client thread in the scheduler's queue, the scheduler uses the weaker WAKEUP_ON_CONFLICT sensitivity level when running it to indicate that the dispatcher may switch among client threads arbitrarily as long as only one client thread attempts to use the CPU at a time.

4.2 Timekeeping and Preemption

The simple FIFO scheduler above can be converted to a round-robin scheduler by introducing some form of clock or timer. For example, if the scheduler is the root scheduler on a CPU, then the scheduler might be directly responsible for servicing clock interrupts. Alternatively, the scheduler may rely on a separate “timer thread” to notify it when a

periodic timer expires. In any case, a timer expiration or clock interrupt is indicated to the scheduler by a message being sent to the scheduler's port. This message causes the scheduler to break out of its `schedule` operation and preempt the CPU from whatever client thread was using it. The scheduler can then move that client to the tail of the ready queue for its priority and give control to the next client thread at the same priority.

4.3 Multiprocessor Support

Since the example scheduler above only contains a single scheduler thread, it can only schedule a single client thread at once. Therefore, although it can be run on a multiprocessor system, it cannot take advantage of multiple processors simultaneously. For example, a separate instance of the FIFO scheduler could be run as the root scheduler on each processor; a client thread assigned to a given scheduler is effectively bound to its scheduler's CPU. Although in some situations this arrangement can be useful, e.g., when each processor is to be dedicated to a particular purpose, in most cases it is not what is needed.

In order for a scheduler to provide "real" multiprocessor scheduling to its clients, where different client threads can be dynamically assigned to different processors on demand, the scheduler itself must be multi-threaded. Assume for now that the scheduler knows how many processors are available, and can bind threads to processors. (This is trivial if the scheduler is run as the root scheduler on some or all processors; we will show later how this requirement can be met for non-root schedulers.) The scheduler creates a separate thread bound to each processor; each of these scheduler threads then selects and runs client threads on that processor. The scheduler threads cooperate with each other using shared variables, e.g., shared run queues in the case of a multiprocessor FIFO scheduler.

Since a scheduler's client threads are supposed to be unaware that they are being scheduled on multiple processors, the scheduler exports only a single port representing the scheduler as a whole to all of its clients. When a client thread wakes up and sends a notification to the scheduler port, the dispatcher arbitrarily wakes up one of the scheduler threads waiting on that port. A good policy is for the dispatcher to wake up the scheduler thread associated with the CPU on which the wakeup is being done; this allows the scheduler to be invoked on the local processor without interfering with other processors unnecessarily. If the woken scheduler thread discovers that the newly woken client should be run on a different processor (e.g., because it is already running a high-priority client but another scheduler thread is running a low-priority client), it can interrupt the other scheduler thread's `schedule` operation by sending it a message or "signal"; this corresponds to sending inter-processor interrupts in traditional systems.

4.3.1 Processor Affinity

Scheduling policies that take processor affinity into consideration[27, 28, 29], can be implemented by treating each scheduler thread as a processor and attempting to schedule a client thread from the same scheduler thread that previously donated CPU time to that client thread. Of course, this will only work if the scheduler threads themselves are consistently run on the same processor. Any processor affinity support in one scheduling layer will only work well if all the layers between it and the root scheduler also consider processor affinity; a mechanism to ensure this is described in the next section.

4.3.2 Scheduler Activations

In the common case, client threads "communicate" with their schedulers implicitly through notifications sent by the dispatcher on behalf of the client threads. However, there is nothing to prevent client threads from *explicitly* communicating with their schedulers through some additional interface. One particularly useful explicit interface is *scheduler activations*[3], which allows clients to determine initially and later track the number of actual processors available to them. Clients can then create or destroy threads as appropriate in order to make use of all available processors without creating superfluous threads that compete with each other uselessly on a single processor.

Since scheduler threads are notified by the dispatcher when a client thread blocks and temporarily cannot use the CPU (e.g., because the thread is waiting for an I/O request or a page fault to be serviced), the scheduler can notify the client in such a situation and give the client an opportunity to create a new thread to make use of the CPU while the original thread is blocked. For example, a client can create a pool of "dormant" threads, or "activations," which the scheduler knows about but normally never runs. If a CPU becomes available, e.g., because of another client thread blocking, the scheduler "activates" one of these dormant threads on the CPU vacated by the blocked client thread. Later, when the blocked thread eventually unblocks and requests CPU time again, the scheduler preempts one of the currently running client threads and notifies the client that it should make one of the active threads dormant again.

Scheduler activations were originally devised to provide better support for application-specific thread packages running in a single user mode process. In an OS kernel that implements CPU inheritance scheduling, extending a scheduler to provide this support should be quite straightforward. However, in a multiprocessor system based on CPU inheritance scheduling, scheduler activations are also useful in stacking of first-class schedulers. As mentioned previously, multiprocessor schedulers need to know the number of processors available in order to use the processors efficiently. As long as a base-level scheduler (e.g., the root scheduler on a set of CPUs) supports scheduler activations, a higher-level multiprocessor scheduler running as a

client of the base-level scheduler can use the scheduler activations interface to track the number of processors available and schedule *its* clients effectively. (Simple single-threaded schedulers that only make use of one CPU at a time don't need scheduler activations and can be stacked on top of any scheduler.)

4.4 Timing

Most scheduling algorithms require a *measurable* notion of time in order to implement preemptive scheduling. For most schedulers, a periodic interrupt is sufficient, although some real-time schedulers may need finer-grained timers whose periods can be changed at each quantum. With CPU inheritance scheduling, the precise nature of the timing mechanism is not important to the general framework; all that is needed is some way for a scheduler thread to be woken up after some amount of time has elapsed. In our implementation, schedulers can register *timeouts* with a central clock interrupt handler; when a timeout occurs, a message is sent to the appropriate scheduler's port, waking up the scheduler. The dispatcher automatically preempts the running thread if necessary and passes control to the scheduler so that it can account for the elapsed time and potentially switch to a different client thread.

4.4.1 CPU Usage Accounting

Besides simply deciding which thread to run next, schedulers often must account for CPU resources consumed. CPU accounting information is used for a variety of purposes, such as reporting usage statistics to the user on demand, modifying scheduling policy based on CPU usage (e.g., dynamically adjusting thread priority), or billing a customer for CPU time consumed for a particular job. As with scheduling policies, there are many possible CPU accounting mechanisms, each with different cost/benefit tradeoffs. The CPU inheritance scheduling framework allows a variety of accounting policies to be implemented by scheduler threads.

There are two well-known approaches to CPU usage accounting: *statistical* and *time stamp-based*[4]. With statistical accounting, the scheduler wakes up on every clock tick, checks the currently running thread, and charges the entire time quantum to that thread. This method is quite efficient, since the scheduler generally wakes up on every clock tick anyway; however, it provides limited accuracy. A variation that provides better accuracy at slightly higher cost is to sample the current thread at random points *between* clock ticks[21]. Alternatively, with time stamp-based accounting, the scheduler reads the current time at every context switch and charges the appropriate thread for the time since the last context switch. This method provides extremely high accuracy, but also imposes a high cost due to lengthened context switch times, especially on systems on which reading the current time is expensive.

In the root scheduler on a processor, these methods can be applied directly. To implement statistical accounting, the scheduler simply checks what thread it ran last upon being woken up by the arrival of a timeout message. To implement time stamp-based accounting, the scheduler reads the current time each time it schedules a different client thread. The scheduler must use the `WAKEUP_ON_BLOCK` sensitivity level in order to ensure that it can check the time at each thread switch and to ensure that idle time is not charged to any thread.

For schedulers stacked on top of other schedulers, CPU accounting becomes a little more complicated because the CPU time supplied to such a scheduler is already "virtual" and cannot be measured accurately by a wall-clock timer. For example, in Figure 3, if scheduler S_1 measures T_2 's CPU usage using a wall-clock timer, then it may mistakenly charge against T_2 time actually used by the high-priority thread T_0 , which S_1 has no knowledge of because it is scheduled by the root scheduler S_0 . In many cases, this inaccuracy caused by stacked schedulers may be ignored in practice on the assumption that high-priority threads and schedulers will consume relatively little CPU time; this assumption is similar to the one made in many existing kernels that hardware interrupt handlers consume little enough CPU time that they may be ignored for accounting purposes. In situations in which this assumption is not valid and accurate CPU accounting is needed for stacked schedulers, virtual CPU time information provided by base-level schedulers can be used instead of wall-clock time, at the cost of additional communication between schedulers. For example, in Figure 3, at each clock tick (for statistical accounting) or each context switch (for time stamp-based accounting), scheduler S_1 could request its own virtual CPU time usage from S_0 instead of checking the current wall-clock time. It then uses this virtual time information to maintain usage statistics for *its* clients, T_1 and T_2 .

4.4.2 Effects of CPU Donation on Timing

As mentioned earlier, CPU donation can occur implicitly as well as explicitly, e.g., to avoid priority inversion when a high-priority thread attempts to lock a resource already held by a low-priority thread. For example, in Figure 5, scheduler S_0 has donated the CPU to high-priority thread T_0 in preference over low-priority thread T_1 . However, it turns out that T_1 is holding a resource needed by T_0 , so T_0 implicitly donates its CPU time to T_1 . Since this donation merely extends the scheduling chain, S_0 is unaware that the switch occurred, and it continues to charge CPU time used to T_0 instead of T_1 which is the thread that is *actually* using the CPU.

While it may seem non-intuitive at first, in practice this is often precisely the desired behavior; it stems from the basic rule that with privilege comes responsibility. While T_0 is donating CPU to T_1 , T_1 is effectively doing work *on behalf of* T_0 , even if T_1 is unaware that it is receiving

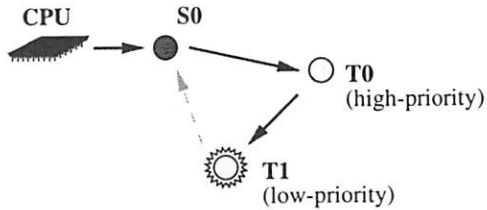


Figure 5: Implicit CPU donation from high-priority thread T_0 to low-priority thread T_1 to avoid priority inversion during a resource conflict.

CPU time from T_0 . Since T_0 's share of the CPU is being used to perform this work, the CPU time consumed must also be charged to T_0 , even if the work is actually being done by another thread. Demonstrated another way, charging T_1 rather than T_0 would allow the accounting system to be subverted. For example, if high-priority CPU time is "expensive" and low-priority CPU time is "cheap," then T_0 could collude with T_1 to use high-priority CPU time while being charged the low-priority "rate" simply by arranging for T_1 to do all the actual work while T_0 blocks on a lock perpetually held by T_1 . This ability to charge the "proper" thread for CPU usage even in the presence of priority inheritance is unnatural and difficult to implement in traditional systems, and therefore is generally *not* implemented by them; on the other hand, this feature falls out of the CPU inheritance framework automatically.

4.5 Threads with Multiple Scheduling Policies

Sometimes it is desirable for a single thread to be associated with two or more scheduling policies at once. For example, a thread may normally run in a real-time rate-monotonic scheduling class; however, if the thread's quantum expires before its work is done, it may be desirable for the thread to drop down to the normal timesharing class instead of simply stopping dead in its tracks.

Support for multiple scheduling policies per thread can be achieved in our framework even under a dispatcher that supports only a single scheduler association per thread, by creating one or more additional "dummy" threads which perpetually donate their CPU time to the "primary" thread. Each of these threads can have a different scheduler association, and the dispatcher automatically ensures that the primary thread always uses the highest priority scheduler available, as described in Section 3.4. In situations in which this solution is not acceptable due to performance or memory overhead, the dispatcher could easily be extended to allow multiple schedulers to be associated with a single thread, so that when such a thread becomes runnable the dispatcher automatically notifies all of the appropriate schedulers.

Although it may at first seem inefficient to notify two or more schedulers when a single thread awakes, in practice many of these notifications never actually need to be *delivered*. For example, if a real-time/timesharing thread wakes

up, finishes all of its work and goes back to sleep again before its real-time scheduling quantum is expired (presumably the common case), then the notification posted to the low-priority timesharing scheduler at wakeup time will be canceled (removed from the queue) when the thread goes to sleep again, so the timesharing scheduler effectively never sees it.

5 Analysis and Experimental Results

We have created a prototype implementation of this scheduling framework and devised a number of tests to evaluate its flexibility, performance, and practicality.

5.1 Test Environment

In order to provide a clean, easily controllable environment, we implemented an initial prototype of this framework in a user-level threads package. The threads package supports common abstractions such as mutexes, condition variables, and message ports for inter-thread communication and synchronization. The package implements separate thread stacks with `set jmp/long jmp`, and the virtual CPU timer alarm signal (`SIGVTALRM`) is used to provide preemption and simulate clock interrupts. We used the virtual CPU timer instead of the wall-clock timer in order to minimize distortion of the results due to other activity in the host Unix system; in a "real" user-level threads package based on this scheduling framework, the normal wall-clock timer would probably be used instead. Our prototype allows threads to wait on only one event at a time; however, there is nothing about the framework that makes it incompatible with thread models in which threads can wait on multiple events at once[25].

Although implemented in user space, our prototype is designed to reflect the structure and execution environment of an actual OS kernel running in privileged mode. For example, the dispatcher itself is passive, nonpreemptible code executed in the context of the currently running thread, an environment similar to that of BSD and other traditional kernels. The dispatcher is cleanly isolated from the rest of the system, and supports scheduling hierarchies of unlimited depth and complexity. Our prototype schedulers are also isolated from each other and from their clients; the various components communicate with each other through message-based protocols that could easily be adapted to operate across protection domains using IPC.

Except where otherwise noted, all measurements were taken on a 100MHz Pentium PC with 32 megabytes of RAM, running FreeBSD 2.1.5.

5.2 Scheduler Configuration

The following experiments are based on scheduling hierarchy shown in Figure 6, which is designed to reflect

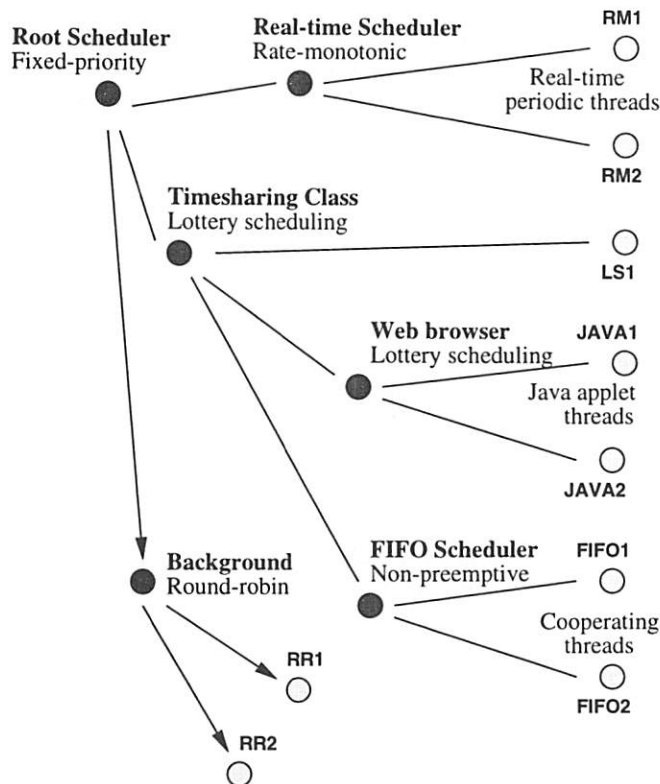


Figure 6: Multilevel scheduling hierarchy used for tests.

the activity present in a general-purpose environment. In this environment, the root scheduler is a nonpreemptive fixed-priority scheduler with a first-come-first-served policy among threads of same priority. This scheduler arbitrates between three scheduling classes: a real-time rate-monotonic scheduler at the highest priority, a lottery scheduler providing a timesharing class, and a simple round-robin scheduler for background jobs. On top of the lottery scheduler managing the timesharing class, an application-specific third-level scheduler manages two threads using lottery scheduling (e.g., a Web browser managing Java applet threads). Finally, a second application-specific scheduler in the timesharing class schedules two cooperating threads using nonpreemptive FIFO scheduling.

5.3 Scheduling Behavior

The purpose of our first experiment is simply to verify that our framework works as expected, producing the same scheduling behavior as traditional single- and multi-class schedulers do in equivalent configurations. Figure 7 shows the scheduling behavior of the threads simulated in our scheduling hierarchy over a time period covering the following sequence of events:

1. A rate-monotonic thread, RM1, becomes runnable and starts consuming CPU time periodically using a fixed reservation of 50%.

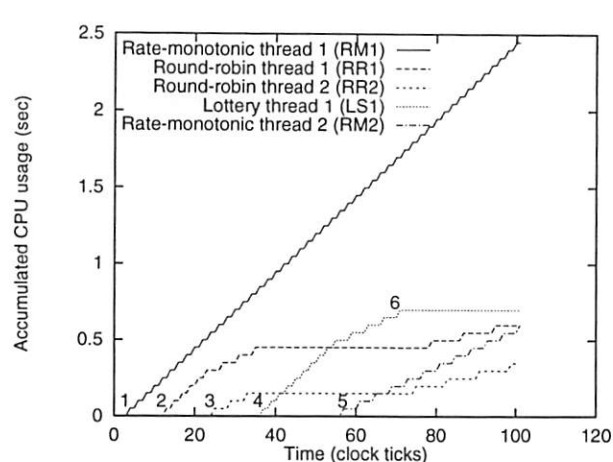


Figure 7: Behavior of a multilevel scheduling hierarchy. Events are numbered according to the description in Section 5.3.

2. A round-robin background thread, RR1, begins a long computation in which it consumes all the CPU time it can get. The rate-monotonic thread, RM1, is unaffected because the fixed-priority root scheduler always schedules the real-time scheduler in preference to the background scheduler.
3. A second round-robin background thread, RR2, becomes runnable at the same priority as RR1.
4. An application thread in the timesharing class becomes runnable, stealing all available CPU time from the background jobs for a short burst of time (e.g., a spreadsheet recalculation).
5. A second rate-monotonic thread, RM2, becomes runnable, consuming 25% of the total CPU time. The time available to the timesharing class is reduced accordingly, but RM1 remains unaffected.
6. The timesharing thread, LS1, finishes its burst of activity, allowing the background jobs to continue once again.

5.4 Modular Control

In order to demonstrate the modular control provided by the framework, we now consider the two third-level schedulers in our example configuration, which represent application-specific schedulers. The first, implementing lottery scheduling, simulates a web browser arbitrating between two downloaded Java applets. The browser can vary the ticket ratio allocated to each applet according to some policy, e.g., giving more CPU time to applets whose output windows are currently on-screen. The second application-level scheduler, a simple non-preemptive FIFO scheduler, represents an application that uses threads merely for programmatic purposes, and does not need to maintain any notion of priority or timeliness between its threads. Using a

non-preemptive scheduler in such an application eliminates unnecessary contention and context switches between application threads without giving up the benefits of preemptive scheduling in other parts of the system.

In this example, the two Java applet threads perpetually consume as much CPU time as possible, and each of the co-operating FIFO threads alternately run for some time and then yield control to the other. Initially, the ticket ratio between the web browser and the cooperative application is 4:1, the ratio between the applet threads is 1:4, and each of the FIFO threads compute for approximately the same time before yielding to the other. Figure 8 shows the behavior of the system across four events:

1. At time 2000, the web browser changes the relative ticket ratio between the Java applet threads to be 1:1, e.g., because the first has come on-screen. The amount of CPU time allocated to the cooperative application remains unchanged, however, demonstrating *load insulation*. Since the timesharing-class scheduler and the web browser's scheduler are both lottery schedulers, this example is equivalent to the use of two *currencies* in a single lottery scheduler[31].
2. The cooperative thread FIFO1 changes its computation so that it now consumes four times the amount of CPU time before yielding to FIFO2. The effective distribution of CPU time changes to 4:1, reflecting the fact that the FIFO scheduling policy makes no attempt at fairness. However, since the timesharing-class scheduler uses a proportional-share policy, the applications are insulated from each other and the web browser is unaffected. This example demonstrates load insulation between *different* scheduling policies.
3. The user changes the ticket allocation between the two applications to 1:1. Both sub-schedulers automatically adjust to the new allocation according to their individual scheduling policies.
4. Finally, the priority of RR1 is raised above that of RR2, causing it to receive all of the CPU time allocated to the round-robin scheduler while leaving the other scheduling domains unaffected.

5.5 Avoidance of Priority Inversion

In the next experiment we demonstrate how priority inversion can be avoided in our framework even among threads running under different scheduling policies. For this experiment, we use rate-monotonic thread RM1, lottery scheduled thread LS1, and background thread RR1. The rate-monotonic thread and the background thread together simulate a data acquisition application in which a high-priority real-time thread periodically receives data

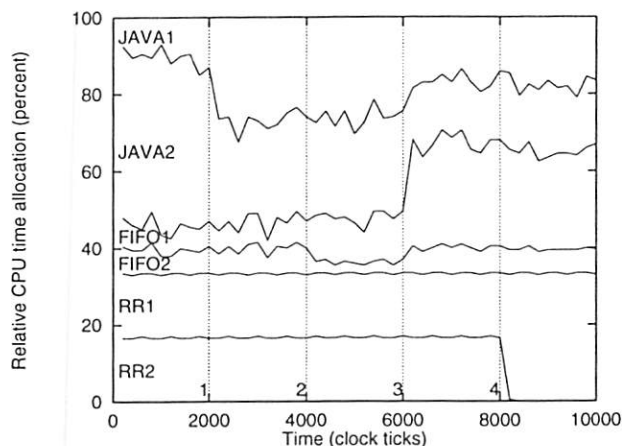


Figure 8: Demonstration of modular control of CPU allocation. Space between the lines represents the percentage of CPU time used by a particular thread. Events occur at 2000-tick intervals, and correspond to the description in Section 5.4.

and stores it in a memory buffer, and a low-priority computational thread inspects that data in the background using spare CPU cycles. In our simulated environment, the high-priority thread wakes up every five clock ticks, acquires a shared mutex lock, and then releases it one clock tick later. The low-priority thread alternately acquires the lock and computes for up to three clock ticks, and then releases it for a varying amount of time. The lottery scheduled thread, LS1, is a medium-priority thread representing other priority inversion-producing activity in the system; it alternately runs and sleeps for random amounts of time up to 20 ticks, but otherwise does not interact with the other threads.

Figure 9 shows a plot of the distribution of latencies experienced by the high-priority thread in attempting to acquire the shared mutex, with and without the voluntary donation protocol described in Section 3.4. When the inheritance protocol is in effect, maximum latency is bounded by the maximum amount of time that the low-priority thread runs with the lock held (three clock ticks), and does *not* reflect the much larger delays that would be imposed if LS1 could prevent RR1 from running while RM1 is waiting for it to release the shared lock. Thus, the framework provides correct real-time behavior even though all three threads run under different scheduling policies: it is not *priority inheritance*, but the more general *CPU inheritance* protocol, that makes this work.

5.6 Multiprocessor Scheduling

To test the ability of our framework to support multiprocessor scheduling, we extended the test environment to simulate a multiprocessor. Since a real multiprocessor machine was not available for this test, we emulated one by time slicing the BSD process at a fine granularity among several “virtual processors.” Each processor has its own

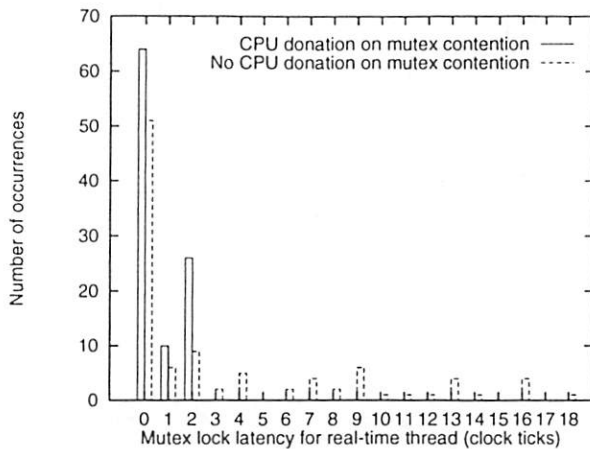


Figure 9: Priority inheritance between schedulers

dispatcher and root scheduler thread, but the root scheduler threads cooperate as described in Section 4.3 to implement a single fixed-priority multiprocessor scheduler with a shared ready queue for each priority level. We tested the multiprocessor scheduler by implementing a simple parallel database lookup application, in which multiple threads repeatedly search for a randomly chosen item in a shared binary tree, locking and unlocking the nodes of the tree as they descend. We ran this application on 1, 2, 4, and 8-processor configurations, with varying numbers of worker threads; in each case, the behavior and performance was exactly as would be expected from a comparable traditional scheduler. Although this experiment demonstrates that the framework extends to multiprocessors, we have not yet explored the effects of stacking multiprocessor scheduling policies, nor have we implemented scheduler activations, which we suspect will be critical to making CPU inheritance scheduling work well in many practical multiprocessor situations.

5.7 Scheduling Overhead

While the preceding experiments demonstrate the basic properties and potential usefulness of our scheduling framework, it remains to be seen whether it is efficient enough in practice. In comparison to traditional multi-class scheduling mechanisms, our framework introduces two additional sources of overhead: first, the overhead caused by the dispatcher itself while computing the thread to switch to after a given event, and second, the overhead resulting from additional context switches to and from scheduler threads. The computation overhead caused by a given scheduling algorithm is not an issue, since the same algorithms can be used in each case.

5.7.1 Dispatcher Costs

To address the first issue, Table 1 shows the basic costs of the dispatcher's computation. These costs are depen-

Scheduling Hierarchy Depth	Dispatch Time (μ S)
Root scheduler only	8.0
2-level scheduling	11.2
3-level scheduling	14.0
4-level scheduling	16.2
8-level scheduling	24.4

Table 1: Basic dispatching costs.

dent on the depth of the scheduling hierarchy since the dispatcher must often iterate through trees and linked lists whose length is proportional to the depth of the hierarchy.

One concern raised by the dependence of dispatch time on scheduling hierarchy depth is that, since in theory the framework imposes no depth limit and the dispatcher itself is always effectively the "highest priority activity in the system," this creates a source of unbounded priority inversion that would be unacceptable in hard real-time systems. However, in practice there is no reason a particular dispatcher must support unlimited depth; the dispatcher in a hard real-time system could limit the depth to four or eight levels, which should be sufficient for all practical purposes and still impose little computational overhead.

5.7.2 Context Switch Costs

The second type of overhead in our framework is the cost of additional context switches to and from scheduler threads. Unfortunately, the cost of a context switch varies widely between different environments, from user-level threads packages in which context switches are almost free, to monolithic kernels in which context switches are several orders of magnitude more expensive. An obvious inference is that our framework is likely to be practical in some environments but not in others. Therefore, in order to gain a meaningful idea of how expensive our framework is likely to be in a given environment, we first measure the *number* of additional context switches produced, which varies with different applications and system loads but is not dependent on thread and protection boundary-crossing costs.

Table 2 shows the context switch statistics observed for each of several example applications: "Client/Server" is an application in which a number of client threads repeatedly invoke services on a smaller set of server threads; "Parallel Database" is the multiprocessor binary search application described in Section 5.6; "Real-time" is the data acquisition application from Section 5.5; finally, "General" is the test in Section 5.3 representing general-purpose computing activity. The chart shows the number of times the dispatcher switched to each thread over the course of the test; user threads are shown separately from scheduler threads. The last row in the table shows the percentage of total context switches that were invocations of scheduler threads. This percentage is consistently near 50% regardless of the application; this indicates that, on average,

	Client/ Server	Parallel Database	Real- time	General
RM1	57		322	101
RM2	19			26
RM3	19			
LS1	25		622	17
JAVA1	46			
FIFO1	9			
RR1	114	238	249	7
RR2	3	242		14
RR3		234		
RR4		243		
User invocations	492	957	1193	165
Root scheduler	262	956	1237	142
Rate monotonic	43		1	65
Lottery scheduler	30		57	3
Java thread scheduler	2			
FIFO scheduler	1			
Round-robin scheduler	8		8	8
Scheduler invocations	346	956	1303	218
Total context switches	838	1913	2496	383
Scheduler invoc. rate	41%	50%	52%	56%

Table 2: Switch Costs for Simple Applications.

approximately one additional scheduler thread invocation can be expected in our framework for each ordinary context switch. However, note that this chart is overly pessimistic because the “scheduler invocations” figures include *all* context switches to scheduler threads, not only those directly related to scheduling: in particular, it includes context switches caused by explicit RPCs, which greatly inflate the counts for the root scheduler in our system because our root scheduler thread also provides message-based timers and other facilities to the rest of the system.

5.7.3 Overhead in Kernel Environments

Although we have not yet implemented our framework in a kernel environment, we can get some idea of how it would perform in such an environment based on the results shown above and a knowledge of how real-world applications exercise the scheduler. Table 3 shows system-wide statistics we gathered for several familiar applications running on FreeBSD 2.1.5, measured using BSD’s `vmstat` command. `gzip` is a compute-intensive application compressing an 8MB file; `gcc` is a build of a 20,000-line program using the GNU C compiler; `tar` represents an I/O-intensive application copying 8MB of source files; and `configure` is an extremely I/O and `fork`-intensive 3000-line Unix shell script. All tests were performed on a networked machine running in multi-user mode with a full complement of daemons but no other user activity, representative of a typical single-user workstation.

For three of the applications in Table 3, `gzip` (best-case), `gcc` (average), and `configure` (worst-case), Figure 10 shows the overall application slowdown that would result if each normal process-to-process context switch was n microseconds more expensive, where n varies from 1

	gzip	gcc	tar	configure
Run time (sec)	26.4	35.3	9.6	26.0
Context switches/sec	11	32	81	202
Traps/sec	10	562	22	3470
System calls/sec	23	651	517	1807
Device interrupts/sec	427	509	3337	1055

Table 3: Scheduling-related statistics for test applications measured under FreeBSD-2.1.5 on a 100MHz Pentium PC.

to 1000 (1 ms). This graph effectively shows the “tolerance” of a system to scheduling overhead in different situations: for example, supposing that typical users would tolerate no more than about 2% slowdown for typical applications such as `gcc`, our framework (or any other scheduling mechanism) would have to add no more than about 300 μ s to the process-to-process context switch time if implemented in FreeBSD. Suppose FreeBSD was changed so that *all* scheduling was done in user mode, adding approximately one additional context switch due to a scheduler invocation for each existing process-to-process context switch. Based on context switch times we measured using the `lmbench` benchmark suite[24], which are approximately 39 μ s on this machine, plus an additional 11 μ s to reflect the dispatcher’s cost (Section 5.7.1), the overall overhead should still be negligible simply because FreeBSD does not context switch all that often (see arrow A on the graph). Furthermore, given BSD’s monolithic design, we would in practice expect at least the root scheduler to be implemented in the kernel and only application-specific schedulers to be in user mode; this would reduce the cost even further.

Of course, microkernels have much less tolerance for scheduling overhead simply because they perform more context switches. For example, in Mach, traps and system calls can be expected to produce about two additional context switches each; for an extremely “purist” microkernel such as L4, in which even device drivers are in user mode, device interrupts would also add an additional two context switches each. Figure 10 also shows a plot of scheduling overhead tolerance for a hypothetical L4-like microkernel, based on the numbers in Table 3 except with traps, system calls, and interrupts each counting as two additional context switches. For example, to keep the overhead under 2% for `gcc` in this system, the additional per-context switch cost must be no more than about 6 μ s (see arrow B), which would be difficult even with L4’s phenomenal 3.2 μ s round-trip RPC time[20]. On the other hand, these “back of the envelope” calculations are pessimistic in several ways: first, an additional scheduler invocation should not be needed for every normal context switch, as explained in Section 5.7.2; second, in a microkernel that dispatches hardware interrupts to threads, we would expect at least a minimal fixed-priority root scheduler to be implemented in the kernel so that device driver threads can be scheduled without the help of user-level schedulers; and

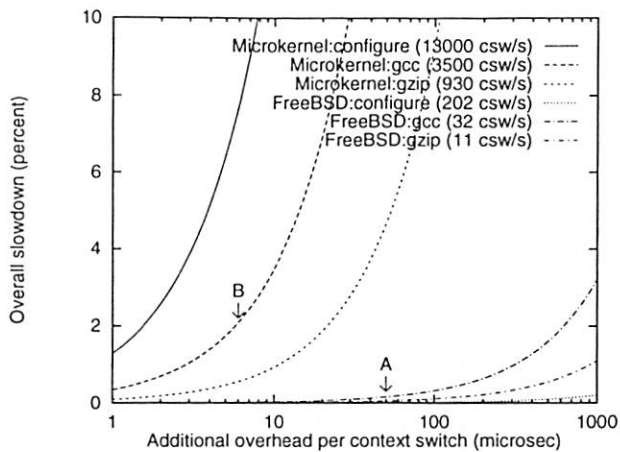


Figure 10: Overall application slowdown as a function of additional overhead per process-to-process context switch. In BSD, only switches between Unix processes count; in Mach, traps and system calls count as RPCs (two context switches each); in L4, device interrupts also count as RPCs.

third, our prototype dispatcher is extremely unoptimized in terms of both its algorithm (it still causes many avoidable context switches) and its implementation (it takes much longer than necessary to compute the next thread to run). There should certainly be many points in the design space at which CPU inheritance scheduling is practical even for microkernels; we are currently working on a second prototype of the framework in our Fluke microkernel[10, 11] in order to evaluate the framework further in this light.

5.8 Code Complexity

As a final useful metric of the practicality of our framework, we measured our prototype's code size and complexity in terms of both raw line count and lines containing semicolons. The entire dispatcher is contained in a single well-commented file of 550 (raw) or 158 (semicolons) lines. Each of our example schedulers is around 200/100 lines long; of course, production-quality schedulers that handle processor affinity, scheduler activations, dynamic priority adjustments, and so on, would probably be significantly bigger.

6 Conclusion

In this paper we have presented a novel processor scheduling framework in which threads are scheduled by other threads. Widely different scheduling policies can co-exist in a single system, providing modular, hierarchical control over CPU usage. Applications as well as the OS can implement customized local scheduling policies, and CPU resources consumed are accounted for and attributed accurately. The framework also cleanly addresses priority inversion by providing a generalized form of priority inheritance that automatically works within and among multiple

diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity and scheduler activations. We have shown that this flexibility can be provided with negligible overhead in environments in which context switches are fast, and that the framework should be practical even in some kernel environments in which context switches are more expensive.

Acknowledgements

For their many thoughtful and detailed comments on earlier drafts we thank the anonymous reviewers and Kevin Jeffay, our shepherd, as well as the members of the Flux project. We are especially grateful to Jay Lepreau for his support and advice, to Kevin Van Maren for considerable help on the results and bibliography, and to Mike Hibler for spending an entire day reviving Bryan's machine after a complete disk failure on Friday the 13th just before the final paper deadline.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.
- [2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation*, pages 127–136, May 1996.
- [3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.
- [4] D. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, Carnegie Mellon University, July 1990.
- [5] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.
- [6] O.-J. Dahl. Hierarchical Program Structures. *Structured Programming*, pages 175–220, 1972.
- [7] S. Davari and L. Sha. Sources of Unbounded Priority Inversions in Real-time Systems and a Comparative Study of Possible Solutions. *ACM Operating Systems Review*, 23(2):110–120, Apr. 1992.

- [8] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.
- [9] R. B. Essick. An Event-based Fair Share Scheduler. In *Proc. of the Winter 1990 USENIX Conf.*, pages 147–161, Washington, D.C., Jan. 1990.
- [10] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ -kernel Environment (draft documents). University of Utah. Postscript and HTML available under <http://www.cs.utah.edu/projects/flux/fluke/html/>, 1996.
- [11] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.
- [12] D. B. Golub. Adding Real-Time Scheduling to the Mach Kernel. Master's thesis, University of Pittsburgh, 1993.
- [13] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as http://java.sun.com/doc/language_environment/.
- [14] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler For Multimedia Operations. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.
- [15] N. Hardy. The KeyKos Architecture. *Operating Systems Review*, Sept. 1985.
- [16] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8), Oct. 1984.
- [17] Institute of Electrical and Electronics, Inc. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Programming Interface (API) — Amendment 1: Realtime Extension [C Language]*, 1994. Std 1003.1b-1993.
- [18] E. D. Jensen. A Benefit Accrual Model of Real-Time. In *Proc. of the 10th IFAC Workshop on Distributed Computer Control Systems*, Sept. 1991.
- [19] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1), Jan. 1988.
- [20] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.
- [21] J. Liedtke. A Short Note on Cheap Fine-grained Time Measurement. *ACM Operating Systems Review*, 30(2):92–94, Apr. 1996.
- [22] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.
- [23] J. M. McKinney. A survey of analytical time-sharing models. *Computing Surveys*, page 105–116, jun 1969.
- [24] L. McVoy and C. Staelin. Imbench: Portable Tools for Performance Analysis. In *Proc. of 1996 USENIX Conf.*, Jan. 1996.
- [25] Microsoft Corporation. *Win32 Programmer's Reference*, 1993. 999 pp.
- [26] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [27] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.
- [28] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, 1995.
- [29] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 26–40, Oct. 1991.
- [30] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.
- [31] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, Nov. 1994. USENIX Assoc.
- [32] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

A Hierarchical CPU Scheduler for Multimedia Operating Systems*

Pawan Goyal, Xingang Guo, and Harrick M. Vin

Distributed Multimedia Computing Laboratory

Department of Computer Sciences, University of Texas at Austin

Taylor Hall 2.124, Austin, Texas 78712-1188

E-mail: {pawang,xguo,vin}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885

URL: <http://www.cs.utexas.edu/users/dmcl>

Abstract

The need for supporting variety of hard and soft real-time, as well as best effort applications in a multimedia computing environment requires an operating system framework that: (1) enables different schedulers to be employed for different application classes, and (2) provides protection between the various classes of applications. We argue that these objectives can be achieved by *hierarchical partitioning* of CPU bandwidth, in which an operating system partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. We present Start-time Fair Queuing (SFQ) algorithm, which enables such hierarchical partitioning. We have implemented a hierarchical scheduler in Solaris 2.4. We describe our implementation, and demonstrate its suitability for multimedia operating systems.

1 Introduction

Over the past few years, computing, communication, and video compression technologies have advanced significantly. Their synergistic advances have made the bandwidth and the storage space requirements of digital video manageable, and thereby have enabled a large class of multimedia applications (e.g., video conferencing, distance learning, news-on-demand services, virtual reality simulation of fire fighting, etc.). Since digital audio and video convey appropriate meaning only when presented continuously in time, such applications impose real-time requirements on the underlying storage, transmission, and processor sub-systems. Specifically, they require an

operating system to allocate resources such as CPU, I/O bus, disk, and network bandwidth in a predictable manner as well as provide Quality of Service (QoS) guarantees (in terms of throughput, response time, etc.). Since no existing operating system meets these requirements, realizing such applications requires conventional operating systems to be extended along several dimensions. Design and implementation of a CPU allocation framework suitable for multimedia operating systems is the subject matter of this paper.

To determine suitable CPU scheduling algorithms, consider the requirements imposed by various application classes that may co-exist in a multimedia system:

- *Hard real-time applications:* These applications require an operating system to deterministically guarantee the delay that may be experienced by various tasks. Conventional schedulers such as the Earliest Deadline First (EDF) and the Rate Monotonic Algorithm (RMA) are suitable for such applications [12].
- *Soft real-time applications:* These applications require an operating system to statistically guarantee QoS parameters such as maximum delay and throughput. Since a large number of such applications are expected to involve video, consider the processing requirements for variable bit rate (VBR) video:

Due to inherent variations in scene complexity as well as the use of intra- and inter-frame compression techniques, processing bandwidth required for compression and decompression of frames of VBR video varies highly at multiple time-scales. For instance, Figure 1 illustrates that the processing bandwidth required for decompressing MPEG video varies from frame-to-frame (i.e., at the time scale of tens of milliseconds) as well as from scene-to-scene (i.e., at the time scale of seconds). Furthermore, these vari-

*This research was supported in part by IBM Graduate Fellowship, IBM Faculty Development Award, Intel, the National Science Foundation (Research Initiation Award CCR-9409666), NASA, Mitsubishi Electric Research Laboratories (MERL), and Sun Microsystems Inc.

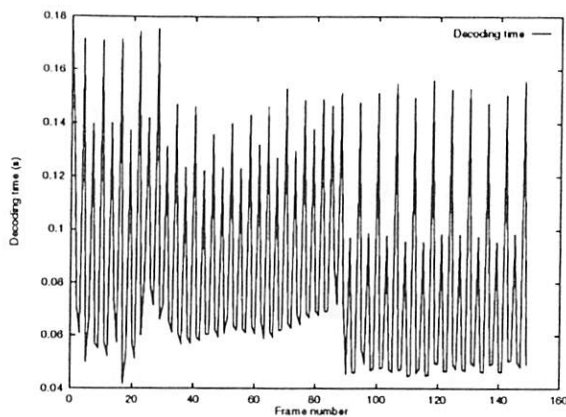


Figure 1 : Variation in decompression times of frames in an MPEG compressed video sequence

ations are unpredictable. These features lead to the following requirements for a scheduling algorithm for VBR video applications:

- Due to the multiple time-scale variations in the computation requirement of video applications, to efficiently utilize CPU, an operating system will be required to over-book CPU bandwidth. Since such over-booking may lead to CPU overload (i.e., cumulative requirement may exceed the processing capacity), a scheduling algorithm must provide some QoS guarantees even in the presence of overload.
- Due to the difficulty in predicting the computation requirements of VBR video applications, a scheduling algorithm must not assume precise knowledge of computation requirements of tasks.

EDF and RMA schedulers do not provide any QoS guarantee when CPU bandwidth is overbooked. Furthermore, their analysis requires the release time, the period, and the computation requirement of each task (thread) to be known a priori. Consequently, although appropriate for hard real-time applications, these algorithms are not suitable for soft real-time multimedia applications. Hence, a new scheduling algorithm that addresses these limitations is desirable.

- *Best-effort applications*: Many conventional applications do not need performance guarantees, but require the CPU to be allocated such that average response time is low while the throughput achieved is high. This is achieved in current systems by time-sharing scheduling algorithms.

From this, we conclude that different scheduling algorithms are suitable for different application classes in a multimedia system. Hence, an operating system framework that enables different schedulers to be employed for different applications is required. In addition to facilitating co-existence, such a framework should provide protection between the various classes of applications. For example, it should ensure that the overbooking of CPU for soft real-time applications does not violate the guarantees of hard real-time applications. Similarly, misbehavior of soft/hard real-time applications, either intentional or due to a programming error, should not lead to starvation of best-effort applications.

The requirements for supporting different scheduling algorithms for different applications as well as protecting application classes from one another leads naturally to the need for *hierarchical partitioning* of CPU bandwidth. Specifically, an operating system should be able to partition the CPU bandwidth among various application classes, and each application class, in turn, should be able to partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. In this paper, we present a flexible framework that achieves this objective.

In our framework, the hierarchical partitioning is specified by a tree. Each thread in the system belongs to exactly one leaf node, and each node in the tree represents either an application class or an aggregation of application classes. Whereas threads are scheduled by leaf node dependent schedulers (determined by the requirements of the application class), intermediate nodes are scheduled by an algorithm that achieves hierarchical partitioning. Specifically, intermediate nodes must be scheduled by an algorithm that: (1) achieves fair distribution¹ of processor bandwidth among competing nodes, (2) does not require a priori knowledge of computational requirements of threads, (3) provides throughput guarantees, and (4) is computationally efficient. We present Start-time Fair Queuing (SFQ) algorithm which meets all of these requirements. We further demonstrate that SFQ is suitable for video applications. We have implemented our hierarchical scheduling framework in Solaris 2.4. We describe our implementation and evaluate its performance. Our results demonstrate that the framework: (1) enables co-existence of heterogeneous schedulers, (2) protects application classes from each other, and (3) does not impose higher overhead than conventional time-sharing schedulers.

Observe that our hierarchical partitioning framework

¹Intuitively, a CPU allocation is fair if, in every time interval, all runnable threads receive the same fraction of CPU bandwidth. This notion of uniform fairness generalizes to weighted fairness when threads have different weights and each thread receives CPU bandwidth in proportion to its weight. We will formalize this notion in Section 3.

also facilitates the development of a QoS manager that allocates resources as per the requirements of applications [10]. To illustrate, if an application requests hard (soft) real-time service, then the QoS manager can use a deterministic (statistical) admission control algorithm which utilizes the capacity allocated to hard (soft) real-time classes to determine if the request can be satisfied, and if so, assign it to the appropriate partition. On the other hand, if an application requests best-effort service, then the QoS manager would not deny the request but assign it to an appropriate partition depending on some other resource sharing policies. A QoS manager can also dynamically change the hierarchical partitioning to reflect the relative importance of various applications. For example, initially soft real-time applications may be allocated very small fraction of the CPU, but when many video decoders requesting soft real-time services are started (possibly as a part of a video conference), the allocation of soft real-time class may be increased significantly. The development of such policies, however, is the subject of future research and beyond the scope of this paper.

The rest of this paper is organized as follows. In Section 2, we introduce our hierarchical CPU scheduling framework. The Start-time Fair Queuing (SFQ) scheduling algorithm and its properties are described in Section 3. The details of our hierarchical SFQ scheduler implementation are described in Section 4. Section 5 describes the results of our experiments. We present related work in Section 6, and summarize our results in Section 7.

2 A Framework for Hierarchical CPU Scheduling

In our framework, the hierarchical partitioning requirements are specified through a *tree* structure. Each thread in the system belongs to exactly one leaf node. Each leaf node represents an aggregation of threads², and hence an application class, in the system. Each non-leaf node in the tree represents an aggregation of application classes. Each node in the tree has a weight that determines the percentage of its parent node's bandwidth that should be allocated to it. Specifically, if r_1, r_2, \dots, r_n denote the weights of the n children of a node, and if B denotes the processor bandwidth allocated to the parent node, then the bandwidth received by node i is given by:

$$B_i = \left(\frac{r_i}{\sum_{j=1}^n r_j} \right) * B$$

Furthermore, each node has a scheduler. Whereas the scheduler of a leaf node schedules all the threads that

²Threads are assumed to be the scheduling entities in the system.

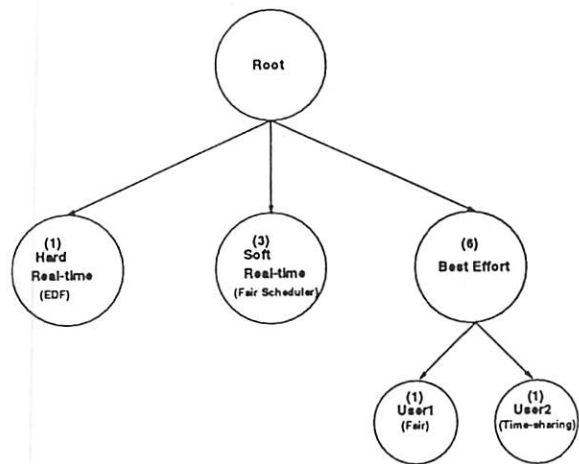


Figure 2 : An example scheduling structure

belong to the leaf node, the scheduler of an intermediate node schedules its child nodes. Given such a scheduling structure, the scheduling of threads occurs hierarchically: the root node schedules one of its child nodes; the child node, in turn, schedules one of its child nodes until a leaf node schedules a thread for execution. Figure 2 illustrates one such scheduling structure. In this example, the root class (node and class are used interchangeably) has three sub-classes: hard real-time, soft real-time and best-effort, with weights 1, 3, and 6, respectively. The bandwidth of the best-effort class has been further divided equally among leaf classes user1 and user2. Furthermore, whereas the soft real-time and user1 leaf classes employ a scheduler that fairly distributes its CPU allocation among its threads, the hard real-time and user2 classes have EDF and time-sharing schedulers, respectively.

Observe that the schedulers at leaf nodes in the hierarchy are determined based on the requirements of the applications. The requirements of a scheduling algorithm for intermediate nodes in the hierarchy, on the other hand, can be defined as follows:

1. To achieve hierarchical partitioning, the algorithm for scheduling intermediate nodes in the hierarchy should:
 - Partition the bandwidth allocated to a class among its sub-classes such that each sub-class gets its specified share.
 - Allocate the residual bandwidth fairly among its sub-classes. For example, in Figure 2, if there are no eligible threads in the hard real-time class, then its allocation should be partitioned between the soft real-time and best-effort nodes in the ratio 3:6.

Both these requirements would be met if the schedul-

ing algorithm partitions the allocation of a class among its sub-classes in proportion to their weights, i.e., achieves weighted fairness. Moreover, as the following example illustrates, a key requirement for such an algorithm is that it should achieve weighted fairness even when the bandwidth available to a class fluctuates over time.

Example 1 Consider the scheduling structure shown in Figure 2. Initially, let there be no threads in the hard and soft real-time classes. Consequently, the best-effort class receives the full CPU bandwidth. When threads join the hard and soft real-time classes, the bandwidth available to the best-effort class goes down to 60% of the CPU bandwidth. In such a scenario, to ensure that user1 and user2 continue to receive equal share of the available bandwidth, the scheduling algorithm for the best-effort class must remain fair even when the available bandwidth fluctuates over time.

2. Since the computational requirements of tasks may not be known precisely, the scheduling algorithm should not assume a priori knowledge of the time duration for which a task executes before it is blocked.
3. To support hard and soft real-time application classes, the scheduling algorithm should provide bounds on minimum throughput and maximum delay observed by nodes. Furthermore, for the bounds to be useful, they should hold in realistic computing environments in which interrupts may be processed at the highest priority [9].
4. To be feasible in general purpose operating systems, the scheduling algorithm should be computationally efficient.

Recently, we have developed a packet scheduling algorithm, referred to as *Start-time Fair Queuing* (SFQ), which achieves fair allocation of network bandwidth [6]. In the next section, we present the algorithm and demonstrate that it meets the above requirements, and hence, is suitable for CPU scheduling in multimedia operating systems.

3 Start-time Fair Queuing

Start-time Fair Queuing (SFQ) is a resource allocation algorithm that can be used for achieving fair CPU allocation. Before we present SFQ, let us formalize the notion of fair allocation. Let r_f be the weight of thread f and $W_f(t_1, t_2)$ be the aggregate work done in interval $[t_1, t_2]$ by the CPU for thread f . For ease of exposition, let the work done by the CPU for a thread be measured by the

number of instructions executed for the thread. Then, a CPU allocation is considered to be fair if, for all intervals $[t_1, t_2]$ in which two threads f and m are runnable, the normalized work (by weight) received by them is identical (i.e., $\frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} = 0$). Clearly, this is an idealized definition of fairness as it assumes that threads can be served in infinitesimally divisible units. Since the threads are scheduled for a quantum at a time, there will be some unfairness. Consequently, the objective of a fair scheduling algorithm is to minimize the resultant unfairness (i.e., ensure that $\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right|$ is as close to 0 as possible)³.

To achieve this objective, SFQ assigns a start tag to each thread and schedules threads in the increasing order of start tags. To define the start tag, let the threads be scheduled for variable length quantum at a time. Also, let q_f^j and l_f^j denote the j^{th} quantum of thread f and its length⁴ (measured in units of instructions), respectively. Let $A(q_f^j)$ denote the time at which the j^{th} quantum is requested. If the thread is making a transition from a blocked mode to runnable mode, then $A(q_f^j)$ is the time at which the transition is made; otherwise it is the time at which its previous quantum finishes. Then SFQ algorithm is defined as follows:

1. When quantum q_f^j is requested by thread f , it is stamped with start tag S_f , computed as:

$$S_f = \max\{v(A(q_f^j)), F_f\} \quad (1)$$

where $v(t)$ is the virtual time at time t and F_f is the finish tag of thread f . F_f is initially 0, and when j^{th} quantum finishes execution it is incremented as:

$$F_f = S_f + \frac{l_f^j}{r_f} \quad (2)$$

where r_f is the weight of thread f .

2. Initially the virtual time is 0. When the CPU is busy, the virtual time at time t , $v(t)$, is defined to be equal to the start tag of the thread in service at time t . On the other hand, when the CPU is idle, $v(t)$ is set to the maximum of finish tag assigned to any thread.
3. Threads are serviced in the increasing order of the start tags; ties are broken arbitrarily.

³ Several other definitions of fairness have been introduced in the networking and operating systems literature. A comparative evaluation of their relative merits, however, is beyond the scope of this paper.

⁴ If l_f^j is the length of quantum q_f^j in terms of instructions, then its time duration is $t_f^j = \frac{l_f^j}{C}$ where C is the rate of execution of the CPU.

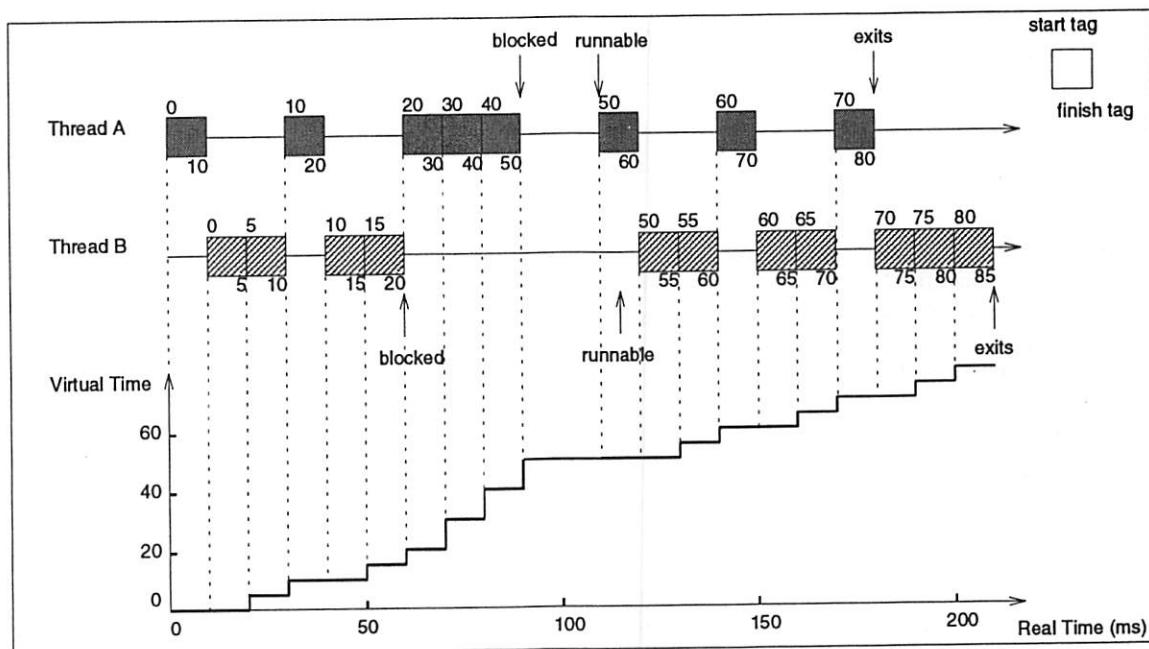


Figure 3 : Computation of virtual time, start tag, and finish tag in SFQ: an example

The following example illustrates the computation of the virtual time, as well as the start and the finish tags (and hence, the process of determining the execution sequence) in SFQ. Consider two threads A and B with weights 1 and 2, respectively, which become runnable at time $t = 0$. Let the scheduling quantum for each thread be 10ms and let $l_f = 10$. Let each thread consume the full length of the quantum each time it is scheduled. Initially, the virtual time $v(t) = 0$. Similarly, the start tags of threads A and B, denoted by S_A and S_B , respectively, are zero (i.e., $S_A = S_B = 0$). Since ties are broken arbitrarily, let us assume, without loss of generality, that thread A is scheduled first for one quantum. Since $v(t)$ is defined to be equal to the start tag of the packet in service, for $0 < t \leq 10$: $v(t) = S_A = 0$. At the end of that quantum, the finish tag of A is computed as $F_A = 0 + \frac{10}{1} = 10$. Moreover, assuming that the thread remains runnable at the end of the quantum, it is stamped with $S_A = \max\{v(10), F_A\} = 10$. At this time, since $S_B < S_A$, the first quantum of thread B is scheduled. Note that since $S_B = 0$, the value of $v(t)$, $10 < t \leq 20$ continues to be equal to 0. At the end of this quantum, the finish tag for B is set to $F_B = 0 + \frac{10}{2} = 5$. Moreover, assuming that B remains runnable at the end of the quantum, we get $S_B = \max\{v(20), F_B\} = 5$. Carrying through this process illustrates that, before thread B blocks at time $t = 60$, threads A and B are scheduled for 20ms and 40ms, respectively, which is in proportion to their weights. When thread B is blocked, the entire CPU bandwidth is avail-

able to thread A, and the value of $v(t)$ changes at the beginning of each quantum of thread A. Now, when thread A blocks at time $t = 90$, the system contains no runnable threads. During this idle period, $v(t)$ is set to $\max\{F_A, F_B\} = \max\{50, 20\} = 50$. When thread A becomes runnable at time $t = 110$, $v(t) = 50$. Hence, thread A is stamped with $S_A = \max\{50, F_A\} = 50$, and is immediately scheduled for execution. On the other hand, when thread B becomes runnable at time $t = 115$, $v(t) = S_A = 50$. Hence, it is stamped with $S_B = \max\{50, F_B\} = \max\{50, 20\} = 50$. From this point, the ratio of CPU allocation goes back to 1:2. Finally, when thread A exits the system, the entire CPU bandwidth becomes available to thread B, until it completes execution. Figure 3 illustrates this complete execution sequence.

3.1 Properties of SFQ

In what follows, we describe the properties of the SFQ scheduling algorithm, and demonstrate that it meets the requirements of a hierarchical scheduler outlined in the previous section.

1. *SFQ achieves fair allocation of CPU regardless of variation in available processing bandwidth and hence meets the key requirement of a scheduling algorithm for hierarchical partitioning. Specifically, in [6], we have shown that regardless of fluctuations in available processor bandwidth, SFQ guarantees that in any interval $[t_1, t_2]$ in which two threads f*

and m are eligible for being scheduled, the following inequality holds:

$$\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m} \quad (3)$$

where l_f^{max} and l_m^{max} , respectively, are the maximum length of quantum for which threads f and m are scheduled⁵. It has been shown in [4] that if a scheduling algorithm schedules threads in terms of quanta and guarantees that $\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right| \leq H(f, m)$ for all intervals, then $H(f, m) \geq \frac{1}{2} \left(\frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m} \right)$. Hence, SFQ is a near-optimal fair scheduling algorithm. In fact, no known algorithm achieves better fairness than SFQ.

2. *SFQ does not require the length of the quantum to be known a priori:* Since SFQ schedules threads in the increasing order of start tags, it does not need the length of the quantum of a thread to be known at the time of scheduling. The length of quantum q_f^j (namely, l_f^j) is required only when it finishes execution, at which time this information is always available. This feature is highly desirable in multimedia computing environments, where the computation requirements are not known precisely and threads may block for I/O even before they are preempted.
3. *SFQ provides bounds on maximum delay incurred and minimum throughput achieved by the threads in a realistic environment:* In most operating systems processing of hardware interrupts occurs at the highest priority. Consequently, the effective bandwidth of CPU fluctuates over time. SFQ provides bounds on delay and throughput even in such an environment. To derive these bounds, however, the variation in the CPU bandwidth has to be quantified. If the maximum rate of occurrence of interrupts and the CPU bandwidth used by the interrupts is known, the effective CPU bandwidth can be modeled as a Fluctuation Constrained (FC) server [11]. A FC server has two parameters; average rate C (instructions/s) and burstiness $\delta(C)$ (instructions). Intuitively, in any interval during a busy period, an FC server does at most $\delta(C)$ less work than an equivalent constant rate server. Formally,

Definition 1 A server is a *Fluctuation Constrained (FC) server* with parameters $(C, \delta(C))$, if for all intervals $[t_1, t_2]$ in a busy period of the server,

⁵ The maximum quantum length may be known a-priori or may be enforced by a scheduler by preempting threads.

the work done by the server, denoted by $W(t_1, t_2)$, satisfies:

$$W(t_1, t_2) \geq C * (t_2 - t_1) - \delta(C) \quad (4)$$

If only bounds on the computation time required by the interrupts is known, then the FC server model is sufficient. However, if distributions of the computation time requirements for processing interrupts are known, then CPU is better modeled as an Exponentially Bounded Fluctuation (EBF) server [11]. An EBF server is a stochastic relaxation of FC server. Intuitively, the probability of work done by an EBF server deviating from the average rate by more than γ , decreases exponentially with γ . Formally,

Definition 2 A server is an *Exponentially Bounded Fluctuation (EBF) server* with parameters $(C, B, \alpha, \delta(C))$, if for all intervals $[t_1, t_2]$ in a busy period of the server, the work done by the server, denoted by $W(t_1, t_2)$, satisfies:

$$P(W(t_1, t_2) < C * (t_2 - t_1) - \delta(C) - \gamma) \leq B e^{-\alpha \gamma} \quad (5)$$

If CPU can be modeled as an FC or EBF server, then SFQ provides throughput and delay guarantees to each of the threads. To determine these guarantees, let the weights of the threads be interpreted as the rate assigned to the threads. For example, a thread that needs 30% of a 100MIPS CPU would have a rate of 30 MIPS. Let Q be the set of threads served by CPU and let $\sum_{n \in Q} r_n \leq C$ where C is the capacity of the CPU. Then, SFQ provides the following throughput and delay guarantees [6]:

Throughput Guarantee: If the CPU is an FC server with parameters $(C, \delta(C))$, then the throughput received by a thread f with weight r_f is also FC with parameters:

$$\left(r_f, r_f \frac{\sum_{n \in Q} l_n^{max}}{C} + r_f \frac{\delta(C)}{C} + l_f^{max} \right) \quad (6)$$

If, on the other hand, the CPU is an EBF server with parameters $(C, B, \alpha, \delta(C))$, then the throughput received by a thread f with weight r_f is also EBF with parameters:

$$\left(r_f, B, \frac{r_f}{C} \alpha, r_f \frac{\sum_{n \in Q} l_n^{max}}{C} + r_f \frac{\delta(C)}{C} + l_f^{max} \right) \quad (7)$$

Hence, if SFQ is used for hierarchical partitioning and if the CPU is an FC(EBF) server, then each of

the sub-classes of the root class are FC(EBF) servers. Using this argument recursively, we conclude that if the CPU is an FC(EBF) server, then each of the sub-classes are also FC(EBF) servers, the parameters of which can be derived using (6) and (7).

Delay Guarantee: If the CPU is a FC server, then SFQ guarantees that the time at which quantum q_f^j of thread f will complete execution, denoted by $L_{SFQ}(q_f^j)$, is given as:

$$L_{SFQ}(q_f^j) \leq EAT(q_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} \quad (8)$$

where $EAT(q_f^j)$ is the expected arrival time of quantum q_f^j . Intuitively, $EAT(q_f^j)$ is the time at which quantum q_f^j would start if only thread f was in the system and the CPU capacity was r_f . Formally,

$$EAT(q_f^j) = \max\{A(q_f^j), EAT(q_f^{j-1}) + \frac{l_f^{j-1}}{r_f}\} \quad (9)$$

where $EAT(q_f^0, r_f^0) = -\infty$.

If the CPU is an EBF server, then SFQ guarantees that $L_{SFQ}(q_f^j)$ is given as follows:

$$P \quad (L_{SFQ}(q_f^j) \leq EAT(q_f^j, r_f^j) + \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} + \frac{\gamma}{C}) \geq 1 - B e^{-\alpha \gamma} \quad 0 \leq \gamma \quad (10)$$

The following example illustrates the delay guarantee of SFQ.

Example 2 Consider a constant rate 100MIPS CPU that serves threads 1, 2, and 3. Let thread 1 reserve 30MIPS and execute 300K instructions every quantum. Also, let the other two threads execute at most 200K instructions every quantum. Then, since $\delta(C) = 0$ for a constant rate CPU, for thread 1, $\sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} + \frac{\delta(C)}{C} = 7ms$. Since executing 300K instructions on a 30MIPS CPU takes 10ms, the expected arrival time of j^{th} quantum of thread 1, assuming it remains runnable at the end of each of its allocated quantum, is $(j-1) * 10ms$. Hence, SFQ guarantees that j^{th} quantum of thread 1 will finish execution by $(j-1) * 10 + 7ms$.

Thus, SFQ not only guarantees fair allocation of CPU to sub-classes, but also provides quantitative bounds on performance.

4. *SFQ is computationally efficient:* Whereas the computation of a start tag only requires one addition and one division, sorting can be efficiently done using a priority queue. The computational complexity of a priority queue is known to $O(\log Q)$, where Q is the number of entities to be scheduled. Since the number of children of a node in a hierarchy are expected to be small (of the order of 2-10), this cost is insignificant when SFQ is used for hierarchical partitioning. Furthermore, no other known algorithm that simultaneously achieves predictable allocation and protection has a lower complexity. Although static priority algorithms have lower complexity, they provide no protection, and hence, have been found to be unsatisfactory for multimedia operating systems [15].

Recall from Section 1 that a scheduling algorithm suitable for video applications should: (1) provide QoS guarantees even in presence of overload, and (2) not require computation requirements to be known precisely. Since SFQ guarantees fair allocation of resources even in presence of overload and does not need computation requirements to be known precisely, it meets these requirements. Hence, it is suitable for video applications as well.

4 Implementation

We have implemented our hierarchical CPU allocation framework in the Solaris 2.4 kernel, which is a derivative of SVR4 UNIX. Our framework utilizes SFQ to schedule all the intermediate nodes for achieving hierarchical partitioning. The requirements of hierarchical partitioning are specified through a tree referred to as a *scheduling structure*. Each node in the scheduling structure has a weight, a start-tag, and a finish-tag that are maintained as per the SFQ algorithm. A non-leaf node maintains a list of child nodes, a list of runnable child nodes sorted by their start-tags, and a virtual time of the node which, as per SFQ, is the minimum of the start-tags of the runnable child nodes. A leaf node has a pointer to a function that is invoked, when it is scheduled by its parent node, to select one of its threads for execution. Each node also has a unique identity and a name similar to a UNIX filename. For example, in the scheduling structure shown in Figure 2, the name of node user1 is "/best-effort/user1". The scheduling structure is created using the following system calls:

- `int hsfq_mknod(char *name, int parent, int weight, int flag, scheduler_id sid)`: This system call creates a node with the given name as a child of node parent in the scheduling structure and returns the identifier of the new node. The flag parameter

identifies the node to be created as a leaf or a non-leaf node. If the node is a leaf node, a pointer to the scheduling function of the class, identified by *sid*, is installed in the node.

- `int hsfq_parse(char* name, int hint)`: This system call takes a name and resolves it to a node identifier in the scheduling structure. The name can be absolute or relative. If it is relative, it is considered to be relative to the node with identifier *hint*.
- `int hsfq_rmnode(int id, int mode)`: This system call is used to remove a node from the scheduling structure. A node can be removed only if it does not have any child nodes.
- `hsfq_move(int from, int to,)`: This system call is used to move a thread from one leaf node to another.
- `hsfq_admin(int node, int cmd, void *args)`: This system call is used for administration operations, other than those mentioned above, on the scheduling structure. Examples of administration operations include changing the weight of a node, determining the weight of a node, etc.

Given a scheduling structure, the actual scheduling of threads occurs recursively. To select a thread for execution, a function `hsfq_schedule()` is invoked. This function traverses the scheduling structure by always selecting the child node with the smallest start tag until a leaf node is selected. When a leaf node is selected, a function that is dependent on the leaf node scheduler, determined through the function pointer that is stored in the leaf node by `hsfq_mknode()`, is invoked to determine the thread to be scheduled. When a thread blocks or is preempted, the finish and the start tags of all the ancestors of the node to which the thread belongs have to be updated. This is done by invoking a function `hsfq_update()` with the duration for which the thread executed and the node identifier of the leaf node as parameters.

A node in the scheduling structure is scheduled if and only if at least one of the leaf nodes in the sub-tree rooted at that node has a runnable thread. The eligibility of a node is determined as follows. When the first thread in a leaf node becomes eligible for scheduling, function `hsfq_setrun()` is invoked with the leaf node's identifier. This function marks the leaf node as runnable and all the other ancestor nodes that may become eligible as a consequence. Note that this function has to traverse the path from the leaf up the tree only until a node that is already runnable is found. On the other hand, when the last thread in a leaf node makes a transition to sleep mode, function `hsfq_sleep()` is called with the leaf node's identifier. This function marks the leaf node as ineligible and

all the other ancestor nodes that may become ineligible as a consequence. This function has to traverse the path from the leaf only until a node that has more than one runnable child nodes is found.

In our implementation, any scheduling algorithm can be used at the leaf node as long as it: (1) provides an interface function that can be invoked by `hsfq_schedule()` to select the next thread for execution, and (2) invokes `hsfq_setrun()`, `hsfq_sleep()`, and `hsfq_update()` as per the rules defined above. We have implemented SFQ as well as modified the existing SVR4 priority based scheduler to operate as a scheduler for a leaf node. The SVR4 leaf scheduler in our implementation, as in the standard release, uses a scheduling algorithm that is dependent on the scheduling class of a thread (e.g., time-sharing, interactive, system, etc.). Hence, in our implementation, a scheduler for a leaf node itself can use multiple scheduling policies.

Observe that the threads in a system may synchronize or communicate with each other, which can result in priority inversion (i.e., a scenario in which a lower priority thread may block the progress of a higher priority thread). The threads that synchronize/communicate may either belong to the same leaf class or different leaf classes. If the threads belong to different leaf classes, the notion of priority inversion is not defined. Furthermore, synchronization between threads belonging to different classes is not desirable, since that may lead to violation of QoS requirements of applications. For example, if a thread in the real-time leaf class synchronizes with a thread in the best-effort class, then, since the best-effort class does not perform any admission control, the QoS requirement of the thread may be violated. Techniques for avoiding priority inversion among threads belonging to the same leaf class, on the other hand, depend on the leaf class scheduler. For example, if the leaf scheduler uses static priority Rate Monotonic algorithm, then standard priority inheritance techniques can be employed [13, 14]. Similarly, when the leaf scheduler is SFQ, priority inversion can be avoided by transferring the weight of the blocked thread to the thread that is blocking it. Such a transfer will ensure that the blocking thread will have a weight (and hence, the CPU allocation) that is at least as large as the weight of the blocked thread.

We envision that our scheduling infrastructure would be used by a QoS manager [10] in a multimedia system (see Figure 4). Applications will specify their QoS requirements to the QoS manager which would: (1) determine the resources needed to meet the QoS requirements of the applications; (2) decide the scheduling class the application should belong to, and create the class if it does not exist; (3) employ class dependent admission control procedures to determine if the resource requirements can be satisfied (some classes may have no admission con-

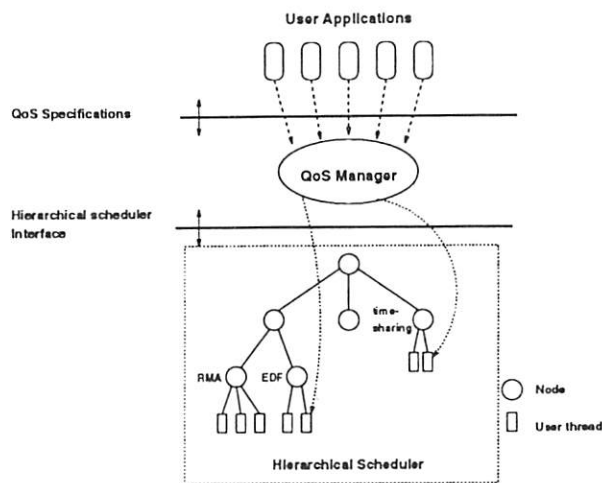


Figure 4 : Quality of Service Manager

trol); and (4) allocate the resources to the application and move it to appropriate class. The QoS manager may also move applications between classes or change the resource allocation in response to change in QoS requirements. It would also dynamically change the relative allocations of different classes so as to effectively meet the requirements of the applications that may coexist at any time. The development of such policies is the subject of future research and beyond the scope of this paper.

5 Experimental Evaluation

We have evaluated the performance of our implementation using a Sun SPARCstation 10 with 32MB RAM. All our experiments were conducted in multiuser mode with all the normal system processes. Most of our experiments were carried out using the Dhrystone V2.1 benchmark, which is a CPU intensive application that executes a number of operations in a loop. The number of loops completed in a fixed duration was used as the performance metric. We evaluated several aspects of the hierarchical scheduler, the results of which are reported in the following sections.

5.1 Limitation of Conventional Schedulers

We had argued that conventional time-sharing schedulers are inadequate for achieving predictable resource allocation in multimedia operating systems. To experimentally validate this claim, we compared the throughput of 5 threads running Dhrystone benchmark under time-sharing and SFQ schedulers. Whereas in the case of SFQ all the threads had equal weight, in the case of time-sharing scheduler all the threads were assigned the same initial user priority. Figure 5 demonstrates that,

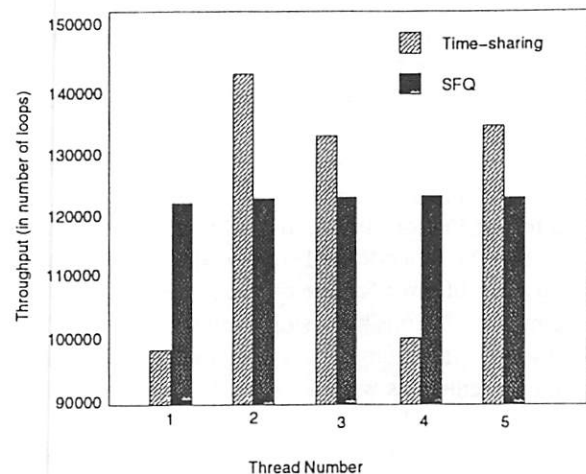


Figure 5 : Comparison of throughput of threads under SFQ and time-sharing schedulers

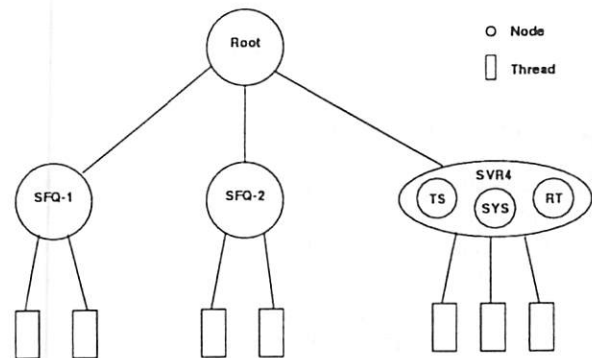


Figure 6 : Scheduling structure used for the experiments

in spite of having the same user priority, the throughput received by the threads in the time-sharing scheduler varies significantly, thereby demonstrating its inadequacy in achieving predictable allocation. In contrast, all the threads in SFQ received the same throughput in conformance with the theoretical predictions. In [15], it has been demonstrated that when a multimedia application is run as a real-time thread in the SVR4 scheduler, the whole system may become unusable. This limitation of the SVR4 scheduler coupled with the unpredictability of time-sharing algorithm clearly demonstrates the need for a predictable scheduling algorithm for multimedia operating system.

5.2 Scheduling Overhead

A key concern in using dynamic priority-based algorithm such as SFQ is that the scheduling overhead may be high. To evaluate the overhead, we determined the ratio of the number of loops completed by a thread in our hierarchical scheduler and the unmodified kernel. In the

hierarchical scheduler, we used the scheduling structure shown in Figure 6 with the threads belonging to node SFQ-1. To determine the effect of the number of threads on the scheduling overhead, the number of threads executing the Dhrystone benchmark was varied from 1 to 20. Figure 7(a) plots the variation in the ratio of the aggregate throughput of threads in our hierarchical scheduler to that in the unmodified kernel against the increase in the number of threads. The ratio was determined by averaging over 20 runs and using a time quantum of 20ms. As Figure 7(a) demonstrates, the throughput achieved by our scheduler is within 1% of the throughput of the unmodified kernel.

To evaluate the impact of the depth of the scheduling structure, the number of nodes between the root class and the SFQ-1 class was varied from 0 to 30. As Figure 7(b) demonstrates, in spite of the significant variation in the depth, the throughput remains within 0.2%. These experiments demonstrate that it is feasible to employ SFQ for hierarchical CPU scheduling.

5.3 Hierarchical CPU allocation

We evaluated the effectiveness of SFQ in achieving hierarchical CPU allocation using the scheduling structure shown in Figure 6. Nodes SFQ-1, SFQ-2 and SVR4 were assigned weights of 2, 6, and 1, respectively. Two threads executing the Dhrystone benchmark were added to leaf nodes SFQ-1 and SFQ-2 (SVR4 node contained all the other threads in the system). Figure 8(a) demonstrates that the aggregate throughput of nodes SFQ-1 and SFQ-2 (measured in terms of number of completed loops of the Dhrystone benchmark) are in the ratio 1:3 (i.e., in accordance to their weights). Observe that due to the variation in the CPU usage of the threads belonging to node SVR4, the aggregate throughput of nodes SFQ-1 and SFQ-2 fluctuates over time. In spite of this variation, nodes 1 and 2 receive throughput in the ratio 1:3, thereby demonstrating the SFQ achieves fair allocation even when the available CPU bandwidth fluctuates over time.

A key advantage of our hierarchical scheduler is that even though different leaf schedulers may be used, each node receives its fair allocation and is isolated from the other nodes. To demonstrate this, we used the scheduling structure shown in Figure 6 with 2 threads in SFQ-1 and 1 thread in SVR4. SFQ-1 as well SVR4 nodes were assigned the same weight. Figure 8(b) demonstrates that the threads in SFQ-1 node as well as SVR4 node make progress and are isolated from each other. Furthermore, both SFQ-1 and SVR4 nodes receive the same throughput. This is in contrast to the standard SVR4 scheduler where a higher priority class, such as the real-time class, can monopolize the CPU.

To demonstrate the feasibility of supporting hard real-time applications in our hierarchical scheduling framework, we used the scheduling structure shown in Figure 6, and executed two threads (namely, thread1 and thread2) in the RT class of the SVR4 node, and an MPEG decoder in SFQ-1 node. The SVR4 and the SFQ-1 nodes were given equal weights. Whereas thread1 executed for 10 ms every 60 ms, thread2 required 150 ms of computation time every 960 ms. Rate monotonic algorithm was used to schedule these two threads. For each thread, a clock interrupt was used to announce the deadline for the current round and the start of a new round of computation. The threads were scheduled for 25ms quanta. We measured the performance of the system using two parameters: (1) *scheduling latency*, which refers to the duration for which a thread has to wait prior to getting access to the CPU after its clock interrupt; and (2) *slack time*, which refers to the difference in time between the deadline and the time at which the current round of computation completes. Figure 9 depicts the variation in scheduling latency and slack time for each round for thread1. Whereas Figure 9(a) illustrates that thread1 gained access to the CPU within a bounded period of time (equal to the length of the scheduling quantum) after its clock interrupt, Figure 9(b) demonstrates that none of the deadlines for thread1 were violated (i.e., the slack time is always positive).

5.4 SFQ as a Leaf Scheduler

To evaluate the use of SFQ as a leaf scheduler, two threads with weights 5 and 10, each running the Berkeley MPEG video player, were assigned to node SFQ-1. Figure 10 plots the number of frames decoded by each thread as a function of time. It demonstrates that the thread with weight 10 decodes twice as many frames as compared to the other thread in any time interval.

5.5 Dynamic Bandwidth Allocation

A QoS manager may dynamically change the bandwidth allocation of classes to meet the application requirements. Hence, SFQ should be able to achieve fair allocation even when bandwidth allocation is dynamically varied. To evaluate this aspect of SFQ, two threads, each executing the Dhrystone benchmark, were run in the SFQ-1 node. The behavior of the threads was varied over time as follows:

- At time 0, both threads were assigned a weight of 4. Hence, the throughput ratio between threads was 4:4.
- At time 4, the weight of thread 2 was changed to 2. Hence, the throughput ratio became 4:2.

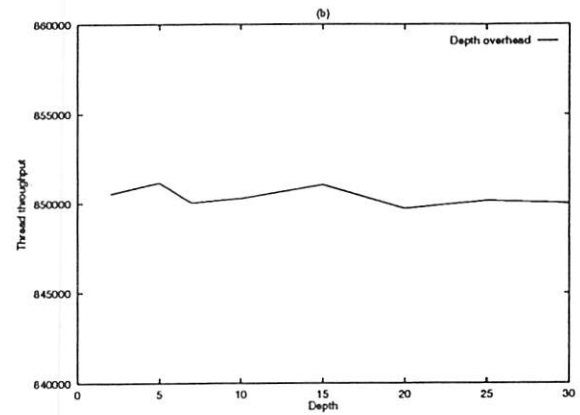
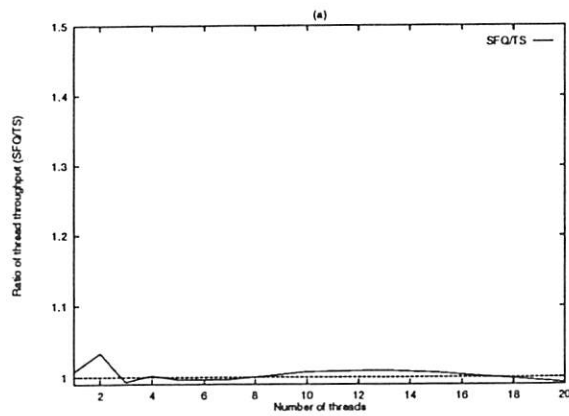


Figure 7 : (a) Ratio of number of loops executed in hierarchical and unmodified scheduler; (b) Variation in throughput with increase in depth of hierarchy

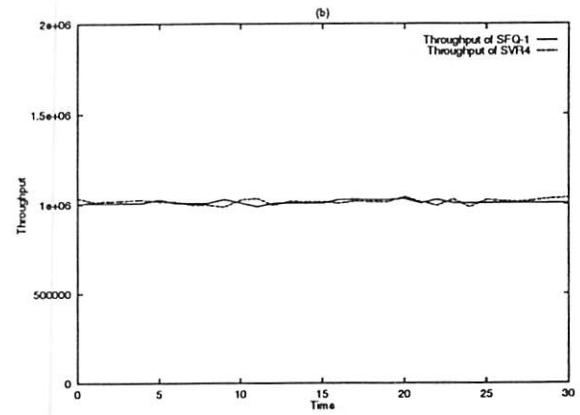
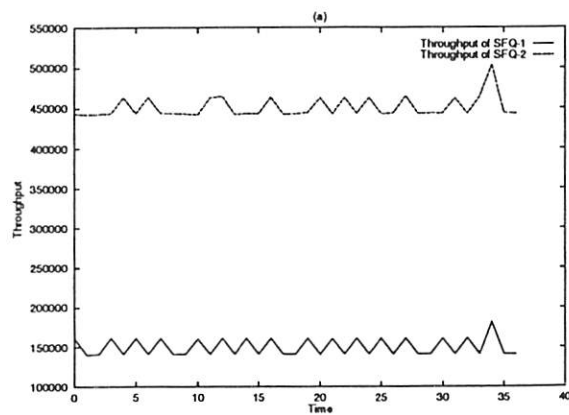


Figure 8 : (a) Aggregate throughput of nodes SFQ-1 and SFQ-2; (b) Throughput of nodes SFQ-1 and SVR4

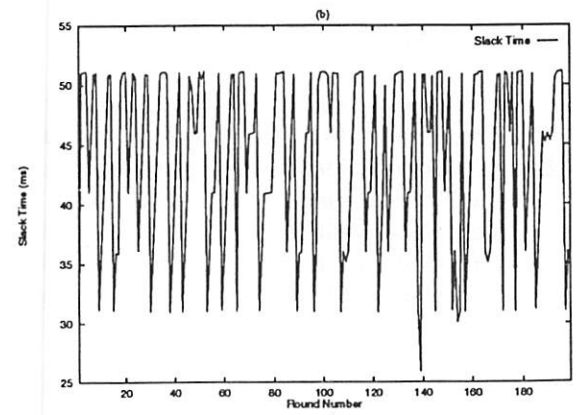
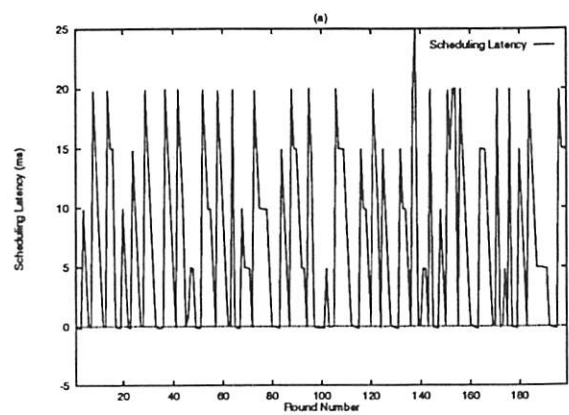


Figure 9 : Variation in: (a) scheduling latency and (b) slack time

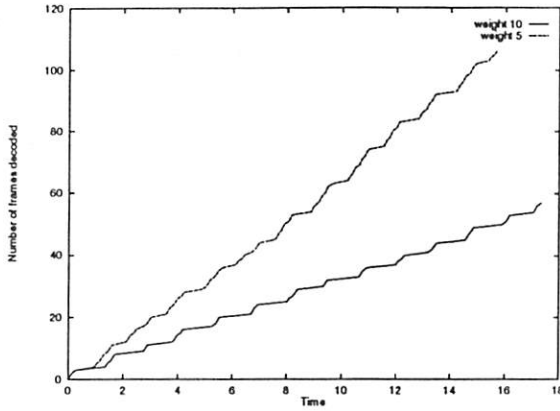


Figure 10 : Number of frames decoded as a function of time

- At time 6, thread 1 was put to sleep. Hence, the throughput ratio became 0:2.
- At time 9, thread 1 resumed execution. Hence, the throughput ratio became 4:2.
- At time 12, the weight of thread 1 was changed to 8. Hence, the throughput ratio became 8:2.
- At time 16, the weight of thread 2 was changed to 4. Hence, the throughput ratio became 8:4.
- At time 22, the weight of thread 1 was changed to 4. Hence, the throughput ratio became 4:4.

Figures 11(a) and 11(b), respectively, illustrate that the throughput of the threads (measured in terms of number of completed loops) and their ratio varies as per the changes in the weights of the threads. This demonstrates that SFQ can achieve fairness even in the presence of dynamic variation in weight assignments.

6 Related Work

We are not aware of any CPU scheduling algorithm that achieves hierarchical partitioning while allowing different schedulers to be used for different applications. However, since a fair scheduling algorithm is the basis for achieving hierarchical partitioning, we discuss other such algorithms proposed in the literature. Most of these algorithms have been proposed for fair allocation of network bandwidth; we have modified their presentation appropriately for CPU scheduling.

The earliest known fair scheduling algorithm is Weighted Fair Queuing (WFQ) [3]. WFQ was designed to emulate a hypothetical weighted round robin server in which the service received by each thread in a round is infinitesimal and proportional to the weight of the thread.

Since threads can only be serviced in quanta at a time, WFQ emulates a hypothetical server by scheduling threads in the increasing order of the finishing times of the quanta of the threads in the hypothetical server. To compute this order, WFQ associates two tags, a *start tag* and a *finish tag*, with every quantum of a thread. Specifically, the start tag $S(q_f^j)$ and the finish tag $F(q_f^j)$ of quantum q_f^j are defined as:

$$S(q_f^j) = \max\{v(A(q_f^j)), F(q_f^{j-1})\} \quad j \geq 1 \quad (11)$$

$$F(q_f^j) = S(q_f^j) + \frac{l_f^{j-1}}{r_f} \quad j \geq 1 \quad (12)$$

where $F(q_f^0) = 0$ and $v(t)$ is defined as the round number that would be in progress at time t in the hypothetical server. Formally, $v(t)$ is defined as:

$$\frac{dv(t)}{dt} = \frac{C}{\sum_{j \in B(t)} r_j} \quad (13)$$

where C is the capacity of the CPU measured in instructions/second and $B(t)$ is the set of runnable threads at time t in the hypothetical server. WFQ then schedules quanta in the increasing order of their finish tags. WFQ has several drawbacks for scheduling a CPU:

- As demonstrated in [6], WFQ does not provide fairness when the processor bandwidth fluctuates over time. Since fairness in the presence of variation in available CPU bandwidth is crucial for supporting hierarchical partitioning, WFQ is unsuitable for a CPU scheduler in a general purpose operating system.
- WFQ requires the length of the quanta to be known a priori. Though the maximum length of the quantum may always be known (as the scheduler can enforce it by preempting a thread), for environments in which the computation requirements are not known precisely, the exact quantum length may not be known. If WFQ assumes the maximum quantum length for scheduling and if the thread uses less than the maximum, the thread will not receive its fair share. On the other hand, if WFQ is modified to reflect the actual length of the execution (by changing the finish tag of a quantum after the end of its execution), then WFQ would have been modified in a non-trivial manner. Though WFQ is known to have bounded fairness, it is not known if the modified algorithm retains its fairness properties.
- WFQ requires the computation of $v(t)$, which, in turn, requires simulation of the hypothetical server. This simulation is known to be computationally expensive [4]. In contrast, SFQ computes the start and the finish tags efficiently.

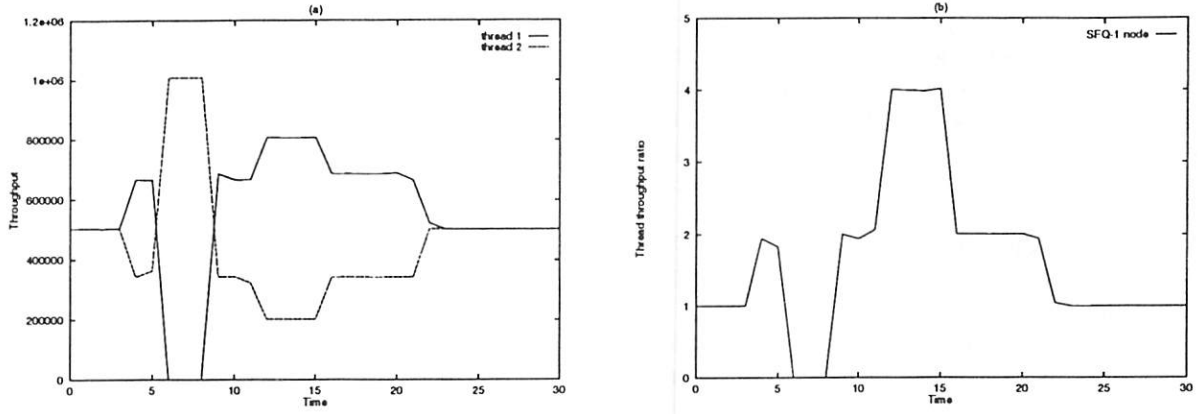


Figure 11 : (a) Throughput of threads 1 and 2; (b) Ratio of throughputs of threads 1 and 2

- The unfairness of WFQ, as derived in [16], is significantly higher than SFQ.
- WFQ provides high delay to low throughput applications. Specifically, it guarantees that quantum q_f^j will complete execution by:

$$EAT(q_f^j) + \frac{l_f^j}{r_f} + \frac{l^{max}}{C} \quad (14)$$

where l^{max} is the maximum quantum length ever scheduled at the CPU. Hence, using (8), we conclude that the difference in maximum delay incurred in SFQ and WFQ, denoted by $\Delta(q_f^j)$, is given as:

$$\Delta(q_f^j) = \sum_{n \in Q \wedge n \neq f} \frac{l_n^{max}}{C} + \frac{l_f^j}{C} - \frac{l_f^j}{r_f} - \frac{l^{max}}{C} \quad (15)$$

Now, if all quanta are of the same lengths, then $\Delta(q_f^j) < 0$ (i.e., SFQ provides a better delay guarantee) if $r_f \leq \frac{1}{|Q|-1}$. Since this condition is expected to hold for low throughput applications, we conclude that SFQ provides lower delay to low throughput applications. Since interactive applications are low throughput in nature, this feature of SFQ is highly desirable for CPU scheduling.

Fair Queuing based on Start-time (FQS) was proposed in [7] to make WFQ suitable for CPU scheduling when quantum length may not be known a priori. It computes the start tag and the finish tag of a quantum exactly as in WFQ. However, instead of scheduling quanta in the increasing order of finish tags, it schedules them in the increasing order of start tags. Since quantum length is not required for computing the start tag, it becomes suitable for CPU scheduling. However, its main drawbacks are that: (1) just as WFQ, it is computationally expensive, and (2) it does not provide fairness when the available

CPU bandwidth fluctuates over time, and consequently it is unsuitable for hierarchical partitioning. Furthermore, it is not known to have any better properties than SFQ.

Self Clocked Fair Queuing (SCFQ), originally proposed in [2] and later analyzed in [4], was designed to reduce the computational complexity of fair scheduling algorithms like WFQ. It achieves efficiency over WFQ by approximating $v(t)$ with the finish tag of the quantum in service at time t . However, since SCFQ also schedules quanta in increasing order of finish tags, it is unsuitable for scheduling CPU in a multimedia operating system. Furthermore, although it has the same fairness and implementation complexity as SFQ, it provides significantly larger delay guarantee than SFQ. Specifically, it increases the maximum delay of quantum q_f^j by $\frac{l_f^j}{r_f}$ [6].

In the OS context, a randomized fair algorithm, termed lottery scheduling, was proposed in [19]. Due to its randomized nature, lottery scheduling achieved fairness only over large time-intervals. This limitation was later addressed by stride scheduling algorithm [18]. The stride scheduling algorithm is a variant of WFQ and, consequently, has all the drawbacks of WFQ. Furthermore, no theoretical properties of the stride scheduling algorithm are known. Recently, a proportionate share resource allocation algorithm, referred to as Earliest Eligible Virtual Deadline First (EEVDF), has been proposed [17].

Hierarchical partitioning of resource allocation was also proposed in [19] using the abstraction of tickets and currencies. In that framework, a thread is allocated tickets in some currency and the currency, in turn, is funded in terms of tickets of some other currency. The “funding” relationship is such that the value of a ticket in every currency can be translated to a value in the base currency. Every thread then is allocated resources in proportion to the value of its tickets in the base currency using lottery scheduling. This achieves hierarchical partitioning since if a thread becomes inactive, the value of the tickets of the

threads that are funded by the same currency increases. This specification of hierarchical partitioning is similar to our scheduling structure. However, the key differences between our framework and the approach of [19] are as follows. First, our framework permits different scheduling algorithms to be employed for different classes of applications, whereas the framework of [19] does not. Second, hierarchical partitioning is achieved in [19] by re-computation of ticket values of every thread that are funded in the same currency or some ancestor of the currency every time a thread gets blocked or exits. This approach not only incurs additional overhead of computing ticket values, but also does not provide any guarantees. Hence, the requirements of hard and soft real-time applications can not be met in this framework. In contrast, our framework achieves hierarchical partitioning through a theoretically sound hierarchical scheduler.

Several other efforts have investigated scheduling techniques for multimedia systems [1, 5, 8, 13]. These scheduling algorithms are complementary to our hierarchical scheduler and can be employed as leaf class scheduler in our framework. Most of these algorithms require precise characterization of resource requirements of a task (such as computation time and period) as well as admission control to achieve predictable allocation of CPU. In contrast, SFQ requires neither of these; it just requires relative importance of tasks (expressed by weights) to be known. It requires admission control only if the applications desire a certain guaranteed minimum CPU bandwidth. Thus, SFQ allows a range of control over CPU allocation: whereas admission control can be used to guarantee a certain minimum CPU allocation to tasks and thus match the performance of the existing algorithms, admission control can be avoided when applications only require relative resource allocation. Such a flexibility is highly desirable in multimedia systems and the lack of it is one of the main disadvantage of existing algorithms. A detailed experimental investigation of the relative merits of these algorithms vis-a-vis SFQ as a leaf class scheduler is the subject of our current research.

7 Concluding Remarks

In this paper, we presented a flexible framework for hierarchical CPU allocation, using which an operating system can partition the CPU bandwidth among various application classes, and each application class, in turn, can partition its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications. We presented the Start-time Fair Queuing (SFQ) algorithm, which enabled such hierarchical partitioning. We demonstrated that SFQ is not only suitable for hierarchical partitioning, but is also suitable for video applications. We have implemented the hierarchical scheduler in Solaris

2.4. We demonstrated that our framework: (1) enables co-existence of heterogeneous schedulers, (2) protects application classes from each other, and (3) does not impose higher overhead than conventional time-sharing schedulers. Thus, our hierarchical scheduling framework is suitable for multimedia operating systems.

References

- [1] D. P. Anderson. Metascheduling for Continuous Media. *ACM Transactions on Computer Systems*, 11(3):266-252, August 1993.
- [2] J. Davin and A. Heybey. A Simulation Study of Fair Queueing and Policy Enforcement. *Computer Communication Review*, 20(5):23-29, October 1990.
- [3] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1-12, September 1989.
- [4] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94*, 1994.
- [5] R. Govindan and D. P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 68-80, October 1991.
- [6] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pages 157-168, August 1996.
- [7] A. Greenberg and N. Madras. How Fair is Fair Queueing. *The Journal of ACM*, 39(3):568-598, July 1992.
- [8] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications*, 15:388-395, July/August 1992.
- [9] K. Jeffay and D.L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium Raleigh-Durham, NC*, pages 212-221, December 1993.
- [10] M.B. Jones, P. Leach, R. Draves, and III J. Barrera. Support for User-Centric Modular Real-Time Resource Management in Rialto Oper-

- ating System. In *Proceedings of NOSSDAV'95, Durham, New Hampshire*, April 1995.
- [11] K. Lee. Performance Bounds in Communication Networks With Variable-Rate Links. In *Proceedings of ACM SIGCOMM'95*, pages 126–136, 1995.
 - [12] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprocessing in a Hard-Real Time Environment. *JACM*, 20:46–61, January 1973.
 - [13] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, May 1994.
 - [14] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the 14th IEEE Real-Time Systems Symp.*, pages 120–130, December 1993.
 - [15] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 41–53, November 1993.
 - [16] D. Stiliadis and A. Varma. Design and Analysis of Frame-based Fair Queueing: A New Traffic Scheduling Algorithm for Packet Switched Networks. In *Proceedings of SIGMETRICS'96*, May 1996.
 - [17] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium (to appear)*, December 1996.
 - [18] C. Waldspurger and W. Wehl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.
 - [19] C. A. Waldspurger and W. E. Wehl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposium on Operating System Design and Implementation*, November 1994.

The Synergy Between Non-blocking Synchronization and Operating System Structure

Michael Greenwald and David Cheriton *
Computer Science Department
Stanford University
Stanford, CA 94305-9040

Abstract

Non-blocking synchronization has significant advantages over blocking synchronization: however, it has not been used to a significant degree in practice. We designed and implemented a multiprocessor operating system kernel and run-time library for high-performance, reliability and modularity. We used non-blocking synchronization, not because it was an objective in itself, but because it became the approach of choice. It was an attractive approach because of the synergy between other structuring techniques we used to achieve our primary goals and the benefits of non-blocking synchronization.

This paper describes this synergy: the structuring techniques we used which facilitated non-blocking synchronization and our experience with this implementation.

1 Introduction

We chose to use non-blocking synchronization in the design and implementation of the Cache Kernel [7] operating system kernel and supporting libraries for several reasons. First, non-blocking synchronization allows synchronized code to be executed in an (asynchronous) signal handler without danger of deadlock. For instance, an asynchronous RPC handler (as described in [24]) can directly store a string into a synchronized data structure such as a hash table even though it may be interrupting another thread updating the same table. With locking, the signal handler could deadlock with this other thread.

Second, non-blocking synchronization minimizes interference between process scheduling and synchronization. For example, the highest priority process can access a synchronized data structure without being delayed or blocked by a lower priority process. In contrast, with blocking synchronization, a low priority process holding a lock can delay a higher priority process, effectively

defeating the process scheduling. Blocking synchronization can also cause one process to be delayed by another lockholding process that has encountered a page fault or a cache miss. The delay here can be hundreds of thousands of cycles in the case of a page fault. This type of interference is particularly unacceptable in an OS like the Cache Kernel where real-time threads are supported and page faults (for non-real-time threads) are handled at the library level. Non-blocking synchronization also minimizes the formation of convoys which arise because several processes are queued up waiting while a single process holding a lock gets delayed.

Finally, non-blocking synchronization provides greater insulation from failures such as fail-stop process(es) failing or aborting and leaving inconsistent data structures. Non-blocking techniques allow only a small window of inconsistency, namely during the atomic compare-and-swap sequence itself. In contrast, with lock-based synchronization the window of inconsistency spans the entire locked critical section. These larger critical sections and complex locking protocols also introduce the danger of deadlock or failure to release locks on certain code paths.

There is a strong synergy between non-blocking synchronization and the design and implementation of the Cache Kernel for performance, modularity and reliability. First, signals are the *only* kernel-supported form of notification, allowing a simple, efficient kernel implementation compared to more complex kernel message primitives, such as those used in V [6]. Class libraries implement higher-level communication like RPC in terms of signals and shared memory regions [24]. Non-blocking synchronization allows efficient library implementation without the overhead of disabling and enabling signals as part of access and without needing to carefully restrict the code executed by signal handlers.

Second, we simplified the kernel and allows specialization of these facilities using the C++ inheritance mechanism by implementing of most operating system mechanisms at the class library level, particularly

*{michaelg, cheriton}@cs.stanford.edu. This work was sponsored in part by ARPA under US Army contract DABT63-91-K-0001. Michael Greenwald was supported by a Rockwell Fellowship.

the object-oriented RPC system [24]. Non-blocking synchronization allows the class library level to be tolerant of user threads being terminated (fail-stopped) in the middle of performing some system library function such as (re)scheduling or handling a page fault.

Finally, the isolation of synchronization from scheduling and thread deletion provided by non-blocking synchronization and the modularity of separate class libraries and user-level implementation of services leads to a more modular and reliable system design than seems feasible by using conventional approaches.

This synergy between non-blocking synchronization and good system design and implementation carries forward in the more detailed aspects of the Cache Kernel implementation. In this paper, we describe aspects of this synergy in some detail and our experience to date.

The main techniques we use for modularity, performance and reliability are atomic DCAS (or Double-Compare-and-Swap), *type-stable memory management* (TSM), and *contention-minimizing data structures* (CMDS).

DCAS (discussed in detail in Section 5) is defined in Figure 1. That is, DCAS atomically updates locations `addr1` and `addr2` to values `new1` and `new2` respectively if `addr1` holds value `old1` and `addr2` holds `old2` when the operation is invoked.

The next section describes type-stable memory management, which facilitates implementing non-blocking synchronization as well as providing several independent benefits to the software structure. Section 3 describes the contention-minimizing data structures which have benefits in performance and reliability for lock-based as well as non-blocking synchronization. Section 4 describes our approach to minimizing the window of inconsistency and the systems benefits of doing so. Section 5 describes the non-blocking synchronization implementation in further detail with comparison to a blocking implementation. Section 6 describes the non-blocking synchronization primitives that we assumed for our approach and a potential hardware implementation. Section 7 describes the performance of our implementation using simulation to show its behavior under high contention. Section 8 describes how our effort relates to previous and current work in this area. We close with a summary of our conclusions and directions for future work.

2 Type-Stable Memory Management (TSM)

Type-stable memory management (TSM) refers to the management of memory allocation and reclamation so that an allocated portion of memory, a *descriptor*, does not change type within some time bound t_{stable} . This is a

```
int DCAS(int *addr1, int *addr2,
         int old1,   int old2,
         int new1,   int new2)
{
    <begin atomic>
    if ((*addr1 == old1) && (*addr2 == old2)) {
        *addr1 = new1; *addr2 = new2;
        return(TRUE);
    } else {
        return(FALSE);
    }
    <end atomic>
}
```

Figure 1: Pseudo-code definition of DCAS (Double-Compare-and-Swap)

fancy name for an extension of an old idea. For example, the process descriptors in many operating systems are statically allocated at system initialization and are thus type-stable for the lifetime of the system execution.

Our notion of TSM incorporates three basic extensions to this conventional type of implementation. First, a descriptor remains a valid instance of the type even when it is not active, i.e. on the free list. Second, TSM allows multiple memory allocation pools for the same type. For example, there can be a pool of thread descriptors per cluster of processors on a large-scale multiprocessor to minimize contention between clusters. Finally, the type of a portion of memory can change over time, but only as long as it is type-stable over some time t_{stable} . More specifically, a descriptor has to be inactive for at least t_{stable} before it can be reallocated as a different type¹. However, for simplicity, we assume an infinite t_{stable} for this discussion.

TSM simplifies the implementation of non-blocking synchronization algorithms. That is, because a descriptor of type T1 is type-stable, a pointer of type T1 * to the descriptor cannot end up pointing to a descriptor of another type as a result of this area of memory being freed and reallocated as type T2.

Consider, for example, the code shown in Figure 2 to do a non-blocking deletion from a linked list².

The delete operation searches down a linked list of descriptors to find the desired element or detect the end of

¹An example of a TSM implementation is a collection of descriptors that are stored in a set of page frames which are allocated and released over time. When more descriptors are required, additional page frames can be allocated from the general pool and when the number of descriptors falls, the descriptors may be consolidated into a smaller number of pages and the excessive page frames returned to the general pool. However, the release of page frames to the general pool must be delayed sufficiently to ensure the type-stability property. This delay provides a useful hysteresis to the movement of pages between this descriptor collection and the general page pool.

²The list is initialized with a dummy node at the head, thus deletion of the first element works correctly.

```

/* Delete elt */
do {
retry:
    backoffIfNeeded();
    version = list->version;

    for (p = list->head;
         (p->next != elt);
         p = p->next) {
        if (p == NULL) { /* Not found */
            if (version != list->version)
                { goto retry; } /* Changed */
            return NULL; /* Really not found */
        }
    }
} while(!DCAS(&(list->version), &(p->next),
              version,          elt,
              version+1,        elt->next));

```

Figure 2: Deletion from the middle of list, protected by DCAS and version number.

the list. If the element is found, the element is atomically deleted from the list by the DCAS operation. The DCAS succeeds only if the list has not been modified since the delete operation started, as determined from the version field.

The code only checks for conflicts once it reaches the desired element or the end of the list. The descriptors are TSM so each pointer is guaranteed to point to a descriptor of this type. Without TSM, the link pointer, *p*, may point to a descriptor that has been deleted and reallocated as a different type. This type error can cause a random bit-field to be interpreted as a pointer, and cause the search to perform incorrectly, raise an exception due to unaligned access, or read a device register. TSM is a simpler and more efficient way of ensuring this type safety than other techniques we are aware of that prevent reallocation (such as automatic garbage collection mechanisms or reference counts), or that detect potential reallocation (such as per-list-element version numbers).

Besides these benefits to non-blocking synchronization, TSM has several important advantages in the construction of modular, reliable, high-performance operating systems. First, TSM is efficient because a type-specific memory allocator can normally allocate an instance of the type faster than a general-purpose allocator can. For example, allocation of a new thread from a free list of (fixed-size) thread descriptors is a simple dequeue operation whereas a general-purpose allocator like `malloc` may have to do a search and subdivision of its memory resources. The class-specific `new` and `delete` operators of C++ support a clean source code representation of TSM. This allocation can be made even more efficient with many types because a free (or inactive) descriptor is already an instance of this type, and so may require less initialization on allocation than a random

portion of memory.

Second, TSM aids reliability because it is easier to audit the memory allocation, locating all the descriptors of a given type and ensuring that pointers that are supposed to point to descriptors of a given type actually do so. With fixed-size descriptors, TSM also avoids fragmentation of memory that arises with general-purpose allocators. Fragmentation can cause failure as well as poor performance. Relatedly, TSM makes it easier to regulate the impact of one type of descriptor on the overall system resources. For example, with a collection of descriptors that are allocated dynamically using the page frame approach described above, the number of pages dedicated to this type can be controlled to avoid exhausting the memory available for other uses, both from overallocation and from fragmentation of memory.

TSM also minimizes the complexity of implementing the caching model [7] of descriptors in the operating system kernel. In this approach, the number of descriptors of a given type is limited but an allocation never fails. Instead, as in a cache, a descriptor is made available by its *dirty* data being written back to the higher-level system management and then reused to satisfy the new allocation request. This mechanism relies on limiting the number of descriptors, being able to locate an allocated descriptor to reclaim, and being able to determine the dependencies on these descriptors. TSM simplifies the code in each of these cases.

TSM also allows a modular implementation. From an object-oriented programming standpoint, there can be a base class descriptor manager class that is specialized to each type of descriptor. For example, there is a `CacheKernelObjMan` class in our operating system kernel that provides the basic TSM allocation mechanism, which is specialized by C++ derivation to implement `Thread`, `AddressSpaceKernel` and `MemMap` types as well as several other types.

3 Data Structures that Minimize Contention

The Cache Kernel was also designed and implemented to minimize both logical and physical contention to provide for efficient non-blocking synchronization. By *logical contention*, we mean contention for access to data structures that need to be controlled to maintain the consistency and semantics of these data structures. By *physical contention*, we mean the contention for access to shared memory that needs to be controlled to maintain the consistency and semantics of the memory system³.

³Physical contention is separate from logical contention because one can have logical contention without physical contention as well as vice versa, so called *false sharing*. For example, if two shared vari-

Minimizing logical contention with non-blocking synchronization minimizes the overhead of conflicting operations failing and being retried. It also avoids the complexity of complex backoff mechanisms as part of the retry.

Most of our techniques for contention minimization are well-known. For example, one aspect of contention minimization is replicating data structures for each processor. In particular, there are per-processor ready and delay queues in the Cache Kernel, so contention on these structures is limited to signal/interrupt handlers and management operations to load balance, etc. being executed by a separate processor.

Similarly, there is a signal delivery cache per processor which allows a significant number of signals to be delivered by a processor without accessing the shared signal mapping data structure, which cannot be made per-processor without replicating the entire structure. This per-processor "cache" approach is similar to that provided by a per-processor TLB for address translation. The TLB reduces access to the real virtual address space mapping structure, which is necessarily shared among threads in the address space.

Contention on a data structure is also reduced in some cases by structuring it as a multi-level hierarchy. For example, a list that is searched frequently may be revised to be a hash table with a version number or lock per bucket. Then, searches and updates are localized to a single bucket portion of the list, reducing the conflict with other operations, assuming they hash to different buckets. The upper levels of the hierarchy are read-only or read-mostly: descriptors are only added at the leaves.

Physical contention is also reduced by using cache-aligned descriptors. TSM with its restricted allocation of descriptors can also reduce the number of pages referenced as part of scan and search operations, reducing the TLB miss rate, another source of physical contention. Finally, in this vein, commonly updated fields are placed contiguously and aligned to hopefully place them in the same cache line, thereby making the updates more efficient.

The *spatial locality* of data access achieved by these techniques provides significant benefit for synchronization, whether non-blocking or conventional locks. This spatial locality also minimizes the consistency overhead when the system is running across multiple processors, with each caching portions of this shared data. In general, our experience (e.g. [10]) suggests that it is better to (re)structure the data structures to reduce contention rather than attempt to improve the behavior of

synchronization techniques under high contention. Low-contention algorithms are simpler and thus easier to get right, and faster as long as contention is actually low.

4 Minimizing the Window of Inconsistency

The Cache Kernel was also structured to minimize the window in which a data structure was inconsistent. This provides *temporal locality* to a critical section. Again, we use familiar techniques. The basic pattern is to read all the values, compute the new values to be written, and then write these new values all at once after verifying that the values read have not changed. Since a structure is generally inconsistent from the time of the first write to the point that the last write completes, removing the computation from this phase minimizes the window of inconsistency. To minimize the cost of verifying that the read values have not changed, we use a version number that covers the data structure and is updated whenever the data structure changes. The use of a version number also avoids keeping track of the actual location read as part of the operation.

The window of inconsistency is also minimized by structuring to minimize physical contention as part of data structure access.

Physical contention increases the time for a processor to perform an operation because it increases the effective memory access time.

These techniques allow efficient non-blocking synchronization. In particular, an update typically consists of a DCAS operation that updates the version number plus one other location, with the version number ensuring that the data structure has not been changed by another concurrent update. That is, the window of inconsistency is reduced to the execution of the DCAS operation itself.

These techniques have other benefits as well. In particular, the reduced window of inconsistency reduces the probability of a failure, such as a thread termination, corrupting the system data structures. They also reduce the complexity of getting critical section code right because it is shorter with fewer separate control paths through it and therefore easier to test. Some of this structuring would be beneficial, if not required, for an implementation using lock-based synchronization because it reduces lock hold time, thereby further reducing contention.

5 Non-Blocking Synchronization Implementation

With the structuring of the Cache Kernel and supporting class libraries described above, non-blocking synchronization is relatively simple to implement. Most data

able can reside in the same cache line unit so there can be physical contention without logical contention if two processor attempt to update the variables simultaneously, each processor updating a separate variable.

structures are collections of fixed-size descriptors. Several collections are queues for service. For example, thread descriptors are queued in the ready queue and a delay queue of their associated processor. Other collections are lookup or search structures such as a hash table with linked list buckets. For example, we organize page descriptors into a lookup structure per address space, supporting virtual-to-physical mapping for the address space.

5.1 The Base Approach

The non-blocking synchronization for these structures follows a common base structure. There is a version number per list. The DCAS primitive is used to atomically perform a write to a descriptor in a list and increment the version number, checking that the previous value of both has not been changed by a conflicting access to the list. Figure 2 illustrated this structure for deleting a descriptor from a list, where the single write to the descriptor was to change the link field of the predecessor descriptor. Inserting a new descriptor D entails initializing D , locating the descriptor in the linked list after which to insert D , writing the D 's link field to point to the next descriptor, and then performing the DCAS to write the link field of this prior descriptor to D and to increment the version, checking both locations for contention as part of the update.

Dequeuing a descriptor from a TSM free list is a degenerate case of deletion because the dequeue always takes place from the head. It is possible to optimize this case and use a single CAS to dequeue without a version number. However, with efficient DCAS support, it is attractive to use DCAS with a version number to allow the version number to count the number of allocations that take place. (As another special case, an operation requiring at most two locations for the reads and writes can be updated directly using DCAS. We have used this approach with array-based stacks and FIFO queues.)

Some operations that involve multiple writes to the same descriptor can be performed by creating a duplicate of this descriptor, performing the modifications and then atomically replacing the old descriptor by the new descriptor if the list has not changed since the duplicate descriptor was created. This approach is a variant of Herlihy's general methodology [13] which can convert a sequential implementation of any data structure into a wait-free, concurrent one. However, we use DCAS to ensure atomicity with respect to the entire data structure (the scope of the version number) even though we are only copying a single descriptor⁴. As a variant of this

⁴The basic Herlihy approach involves copying the entire data structure, modifying the copy, and then atomically replacing the old copy with the new copy using CAS, and retrying the entire copy and modifying if there is a conflict. Our approach reduces the allocation and copy cost to a single descriptor rather than the entire data structure but requires DCAS.

approach, the code can duplicate just a portion of the descriptor, update it and use DCAS to insert it in place of the original while updating a version number. If a thread fails before completing the insertion, we rely on a TSM-based audit to reclaim the partially initialized descriptor after it is unclaimed for t_{stable} time.

As a further optimization, some data structures allow a descriptor to be removed, modified and then reinserted as long as the deletion and the reinsertion are each done atomically. This optimization saves the cost of allocating and freeing a new descriptor compared to the previous approach. This approach requires that other operations can tolerate the inconsistency of this descriptor not being in the list for some period of time. For example, the Cache Kernel signal delivery relies on a list of threads to which a signal should be delivered. A thread fails to get the signal if it is not in the list at the time a signal is generated. However, we defined signal delivery to be best-effort because there are (other) reasons for signal drop so having signal delivery fail to a thread during an update is not a violation of the signal delivery semantics. Programming the higher-level software with best-effort signal delivery has required incorporating timeout and retry mechanisms but these are required for distributed operation in any case and do not add significant overhead [24]. These techniques, related to the transport-layer in network protocols, also make the system more resilient to faults.

Note that just having a search mechanism retry a search when it fails in conjunction with this approach can lead to deadlock. For example, if a signal handler that attempts to access descriptor D , retrying until successful, is called on the stack of a thread that has removed D to perform an update, the signal handler effectively deadlocks with the thread.

5.2 Dealing with Multiple Lists

A descriptor that is supposed to be on multiple lists simultaneously complicates these procedures. So far, we have found it feasible to program so that a descriptor can be in a subset of the lists, and inserted or deleted in each list atomically as separate operations. In particular, all the data structures that allow a descriptor to be absent from a list allow the descriptor to be inserted incrementally.

Overall, the major Cache Kernel [7] data structures are synchronized in a straightforward manner. Threads are in two linked lists: the ready queue and the delay queue. Descriptor free lists are operated as stacks, making allocation and deallocation simple and inexpensive. The virtual to physical page maps are stored in a tree of depth 3 with widths of 128, 128, and 64 respectively. Although the 128 immediate descendants of the root are never deleted, sub-trees below them can be unloaded. Modifications to a map on level 3 are synchronized using DCAS with its parent's version number to make sure that the entire

subtree has not been modified in conflict with this update. Finally, the Cache Kernel maintains a "dependency map" that records dependencies between objects, including physical to virtual mappings. It is implemented as a fixed-size hash table with linked lists in each bucket. The signal mapping cache structure, (an optimization for signal delivery to active threads), is also a direct mapped hash table with linked lists in each bucket. The majority of uses of single CAS are for audit and counters.

Synchronization of more complex data structures than we have encountered can be handled by each operation allocating, initializing and enqueueing a "message" for a server process that serially executes the requested operations. Read-only operations can still proceed as before, relying on a version number incremented by the server process. Moreover, the server process can run at high priority, and include code to back out of an operation on a page fault and therefore not really block the operation anymore than if the operation was executed directly by the requesting process. The server process can also be carefully protected against failure so the data structure is protected against fail-stop behavior of a random application thread, which may be destroyed by the application.

This approach was used by Pu and Massalin [17]. For example, a general-purpose memory page allocator can be synchronized in this manner, relying on a TSM memory pool to minimize the access to the general allocator. However, in our code to date, the only case of queueing messages for a server module arises with device I/O. This structure avoids waiting for the device I/O to complete and is not motivated by synchronization issues.

Other work has investigated other alternatives or optimizations of this approach, in which helper functions are executed by a new thread if there is work left to complete or rollback by a previous thread accessing this data structure. For example, Israeli et al. [16] describe a non-blocking heap implemented using 2-word LL/SC along these lines, performing multiple updates as multiple distinct operations. However, to date, we have not needed to employ these so-called *helper* techniques and therefore cannot comment on their actual practicality or utility. Moreover, it seems questionable from a reliability standpoint to have threads from separate address spaces sharing access to complex data structures. These data structures are also more difficult to program and to maintain and often provide marginal performance benefits in practice, particularly when synchronization overhead is taken into account. Their asymptotic performance benefits are often not realized at the scale of typical operating system data structures.

5.3 Comparison to Blocking Synchronization

Much of the structuring we have described would be needed, or at least beneficial, even if the software used blocking synchronization. For instance, TSM has a strong set of benefits as well as contributing to the other techniques for minimizing contention and reducing the window of inconsistency.

We have found that the programming complexity of non-blocking synchronization is similar to conventional blocking synchronization. This differs from the experience of programmers using CAS-only systems. DCAS plays a significant part in the complexity reduction. Using the crude metric of lines of code, a CAS implementation (Valois) of concurrent insertion/deletion from a linked list requires 110 lines, while the corresponding DCAS implementation requires 38 (a non-concurrent DCAS implementation takes 25). The CAS-only implementation of a FIFO queue described in [18] requires 37 lines, our DCAS version only 24. The DCAS versions are correspondingly simpler to understand and to informally verify as correct. In many cases, using DCAS, the translation from a well-understood blocking implementation to a non-blocking one is straightforward. In the simple case described in Figure 2, the initial read of the version number replaces acquiring the lock and the DCAS replaces releasing the lock.

In fact, version numbers are analogous to locks in many ways. A version number has a *scope* over some shared data structure and controls contention on that data structure just like a lock. The scope of the version number should be chosen so that the degree of concurrency is balanced by the synchronization costs. (The degree of concurrency is usually bounded by memory contention concerns in any case). Deciding the scope of a version number is similar to deciding on the granularity of locking: the finer the granularity the more concurrency but the higher the costs incurred. However, a version number is only modified if the data structure is modified whereas a lock is always changed. Given the frequency of read-only operations and the costs of writeback of dirty cache lines, using read-only synchronization for read-only operations is attractive. Finally, version numbers count the number of times that a data structure is modified over time, a useful and sometimes necessary statistic.

Finally, the overall system complexity using blocking synchronization appears to be higher, given the code required to get around the problems it introduces compared to non-blocking synchronization. In particular, special coding is required for signal handlers to avoid deadlock. Special mechanisms in the thread scheduler are required to avoid the priority inversion that locks can produce. And, additional code complexity is required to achieve reliable operation when a thread can be terminated at a

random time. For example, some operations may have to be implemented in a separate server process.

A primary concern with non-blocking synchronization is excessive retries because of contending operations. However, our structuring has reduced the probability of contention and the conditional load mechanism described in the next section can be used to achieve behavior similar to lock-based synchronization.

6 Non-blocking Synchronization Primitives

Our approach assumes an efficient implementation of DCAS functionality. In this section, we briefly outline an instruction set extension to the load-linked/store-conditional instructions to support DCAS. (A software implementation is discussed in Section 6.1.) With a processor supporting load-linked (LL) and store-conditional (SC) instructions, add two instructions:

1. LLP (load-linked-pipelined): load and link to a second address after a LL. This load is linked to the following SCP.
2. SCP (store-conditional-pipelined): Store to the specified location provided that no modifications have been made to either of the memory cells designated by *either* of the most recent LL and LLP instructions and these cache lines have not been invalidated in the cache of the processor performing the SCP.

If a LLP/SCP sequence nested within an LL/SC pair fails, the outer LL/SC pair fails too.

DCAS is then implemented by the instruction sequence shown in Figure 3 (using R4000 instructions in addition to the LL/SC(P) instructions). The LL and LLP instructions in lines 1 and 2 “link” the loads with the respective stores issued by the following SC and SCP instructions. Lines 3 and 4 verify that (T0) and (T1) contain V0 and V1, respectively. The SCP and SC in lines 5 and 6 are conditional. They will not issue the stores unless (T0) and (T1) have been unchanged since lines 1 and 2. This guarantees that the results of CAS in lines 3 and 4 are still valid at line 6, or else the SC fails. Further, the store issued by a successful SCP is buffered pending a successful SC. Thus, SC in line 6 writes U1 and U0 to (T1) and (T0) atomically with the comparison to V0 and V1⁵.

⁵Given data structures that are protected by a version number, i.e. DCAS is actually a Compare-And-Double-Swap (CADS) — the second value cannot have changed if the version number is unchanged. In these cases a minor optimization is possible and line 4 can be deleted.

```
/*
 * If (T0) == V0, and (T1) == V1, then
 * atomically store U0 and U1 in T0 and T1
 */
DCAS(T0, T1, V0, V1, U0, U1)
;; Get contents of addresses in registers.
1 LL      T3, (T1)
2 LLP     T2, (T0)
;; Compare to V0 and V1. If unequal, fail.
3 BNE     T2, V0, FAIL
4 BNE     T3, V1, FAIL
;; If equal, and unchanged since LOAD,
;; store new values
5 SCP     U0, (T0)
6 SC      U1, (T1)
;; Success of SC and SCP is stored in U1
BLEZ     U1, FAIL
...
```

Figure 3: DCAS Implementation using LL/SC and LLP/SCP. Success or failure of SC (and thus of the DCAS operation) is returned in U1 or whatever general register holds the argument to SC. 1 denotes success, 0 failure. If the next instruction tries to read U1, the hardware interlocks (as it already does for LL/SC) if the result of SC is not already in U1.

We have worked out a detailed design for the implementation of these two instructions in a RISC processor such as the R4000 but the description is omitted for brevity.

6.1 Software Implementation of DCAS

DCAS functionality can be implemented in software using a technique introduced by Bershada [4]. DCAS is implemented using a lock known to the operating system. If a process holding this lock is *delayed* by a context switch, the operating system rolls back the process out of the DCAS procedure and releases the lock. The rollback procedure is relatively simple because the DCAS implementation is simple and known to the operating system. Moreover, the probability of a context switch in the middle of the DCAS procedure is low because it is so short, typically a few instructions. Thus, the rollback cost is incurred infrequently.

This technique can be used more generally to implement other primitives such as n-location CAS. We focus on DCAS implementation because the primary relation to our work is offering a software implementation of DCAS as an alternative to our proposed hardware support. It also seems simpler to just implement rollback for DCAS compared to more general primitives.

This approach has the key advantage of not requiring hardware extensions over the facilities in existing systems. Moreover, its performance may be comparable to our hardware extensions, especially on single processors or small-scale multiprocessors. Further measurements are required here. However, there are a few concerns.

First, there is the cost of locking. The straight-forward implementation requires the DCAS procedure to access a common global lock from all processes. In a multi-level memory with locks in memory, the memory contention between processors for this lock can be significant. For example, the data structure may be in a shared segment that is mapped in by two independent processes. If the locks are associated with each DCAS instance, there is more cost and complexity to designate the locks and critical section to the operating system and to implement the rollback. The locking and unlocking also modifies the cache line containing the lock, further increasing the cost of this operation because writeback is required.

Second, Bershad's approach requires rereading the two locations from memory as well as an extra read and write to set the lock and write to clear the lock.

Third, on multiprocessors, care must be used by readers of shared data structures if they want to support unsynchronized reads. Without depending on the lock, readers can see intermediate states of the DCAS, and read tentative values that are part of a DCAS that fails. Requiring synchronization for reads significantly increases contention on the global lock. Note that in many cases TSM reduces the danger of unsynchronized reads because the reads cannot cause type errors. Writes are protected by the global lock, and the final DCAS will detect that the unsynchronized reads were suspect, and fail. Systems that provide hardware DCAS require no additional read synchronization beyond that performed automatically by the memory system. Further experience and measurements are required to determine whether this is a significant issue on real systems.

Finally, the Bershad mechanism seems harder to test under all conditions. For instance, it is possible that one of the write operations that the rollback needs to undo is to an area of memory that has been paged out or that one of the addresses is illegal. The system also needs to ensure that a thread is rolled back out of any DCAS critical section if it is terminated. We believe our hardware implementation is simpler to verify and naturally operates on top of the virtual memory management of the system and on top of directly accessible physical memory at the lowest level of the system software. It is of concern that a minor change to the software mechanisms in Bershad's scheme could result in very subtle errors in execution that could go undetected in a system for a long period of time.

6.2 Hardware Contention Control

As a further extension, a processor can provide a conditional load instruction or `Cload`. The `Cload` instruction is a load instruction that succeeds only if the location being loaded does not have an *advisory lock* set on it, setting the advisory lock when it does succeed.

With `Cload` available, the version number is loaded

initially using `Cload` rather than a normal load. If the `Cload` operation fails, the thread waits and retries, up to some maximum, and then uses the normal load instruction and proceeds. This waiting avoids performing the update concurrently with another process updating the same data structure. It also prevents potential starvation when one operation takes significantly longer than other operations, causing these other frequently occurring operations to perpetually abort the former. It appears particularly beneficial in large-scale shared memory systems where the time to complete a DCAS-governed operation can be significantly extended by wait times on memory because of contention, increasing the exposure time for another process to perform an interfering operation. Memory references that miss can take 100 times as long, or more, because of contention misses. Without `Cload`, a process can significantly delay the execution of another process by faulting in the data being used by the other process and possibly causing its DCAS to fail as well.

The cost of using `Cload` in the common case is simply testing whether the `Cload` succeeded, given that a load of the version number is required in any case.

`Cload` can be implemented using the cache-based advisory locking mechanism implemented in ParaDiGM [8]. Briefly, the processor advises the cache controller that a particular cache line is "locked". Normal loads and stores ignore the lock bit, but the `Cload` instruction tests and sets the cache-level lock for a given cache line or else fails if it is already set. A store operation clears the bit. This implementation costs an extra 3 bits of cache tags per cache line plus some logic in the cache controller. Judging by our experience with ParaDiGM, `Cload` is quite feasible to implement.

7 Performance

The performance on the ParaDiGM experimental multiprocessor is first discussed. We then discuss results from simulation indicating the performance of our approach under high contention. Finally, we discuss aspects of overall system performance.

7.1 Experimental Implementation

The operating system kernel and class libraries run on the ParaDiGM architecture [8]. The basic configuration consists of 4-processor Motorola 68040-based multiprocessors running with 25 MHz clocks. The 68040 processor has a DCAS instruction, namely CAS2. This software also runs with no change except for a software implementation of DCAS, on a uniprocessor 66 MHz PowerPC 603. We have not implemented it on a multiprocessor PowerPC-based system to date.

Kernel synchronization uses DCAS in 27% of the critical sections and otherwise CAS. However, the DCAS

uses are performance-critical, e.g. insert and deletion for key queues such as the ready queue and delay queue. The only case of blocking synchronization is on machine startup, to allow Processor 0 to complete initialization before the other processors start execution.

The overhead for non-blocking synchronization is minimal in extra instructions. For example, deletion from a priority queue imposes a synchronization overhead of 4 instructions compared to no synchronization whatsoever, including instructions to access the version number, test for DCAS success and retry the operation if necessary. This instruction overhead is comparable to that required for locked synchronization, given that lock access can fail thus requiring test for success and retry.

The Motorola 68040's CAS2 [25] is slow, apparently because of inefficient handling of the on-chip cache so synchronization takes about 3.5 microseconds in processor time. In comparison, spin locks take on average 2.1 μ secs and queuelocks take about 3.4 μ secs. In contrast, the extended instructions we propose in Section 6 would provide performance comparable to any locking implementation. In particular, it requires 16 extra instructions (including the required no-ops) plus an implicit SYNC in an R4000-like processor. A careful implementation would allow all instructions other than the SYNC to execute at normal memory speed. The performance would then be comparable to the roughly 24 instruction times required by the R4000 lock/unlock sequence. Figure 4 compares the overhead in terms of instruction times.

Operation	Instruction Times
DCAS using CAS2 on 68040	114
DCAS using LLP/SCP	26
SGI R3000 lock/unlock	70
R4000 lock/unlock	24

Figure 4: Approximate instruction times of extra overhead to synchronize deletion from a priority queue. This overhead does not include the backoff computation.

7.2 Simulation-Based Evaluation

The actual contention for the kernel data structures in our current implementation is low and we did not have the ability to create high contention at the time of writing.

To understand how our system behaves under heavy load, we have simulated insertion/deletion into a singly linked list under loads far heavier than would ever be encountered in the Cache Kernel.

Our simulation was run on the Proteus simulator [5], simulating 16 processors, a cache with 2 lines per set, a shared bus, and using the Goodman cache-coherence protocol. All times are reported in cycles from start of test until the last processor finishes executing. Memory la-

tency is modeled at 10 times the cost of a cache reference. The cost of a DCAS is modeled at 17 extra cycles above the costs of the necessary memory references. The additional cost of a CAS over an unsynchronized instruction referencing shared memory is 9 cycles.

Four algorithms were simulated:

1. *DCAS/Cloud*: Our DCAS algorithm with contention controlled by advisory locking, as implemented on Paradigm.
2. *DCAS/A&F*: DCAS algorithm with contention controlled by OS intervention as proposed by Allemany and Felten [1] and described in Section 8.4.
3. *CAS*: An implementation using only CAS and supporting a much higher degree of concurrency based on a technique by Valois [23] ⁶.
4. *SpinLock*: Spin-lock with exponential back-off as a base case.

Each test performed a total of 10,000 insertions and deletions, divided evenly between all processes. We varied the number of processors from 1 to 16 and the number of processes per processor from 1 to 3. We also controlled the rate of access to the list by each process by doing local "work" between the insertion and deletion. The work varied from 20 to 2000 cycles.

These simulations indicate that the Cache Kernel DCAS algorithms perform as well or better than CAS or spin locks.

Figure 5 shows the performance with 1 process per processor, and minimal work between updates. The basic cost of 10,000 updates is shown at $N = 1$, where all accesses are serialized and there is no synchronization contention or bus contention. At $N = 1$, cache contention due to collisions is small, the hit rate in the cache was over 99% in all algorithms. At more than one processor, even assuming no synchronization contention and no bus contention, completion time is significantly larger because the objects must migrate from the cache of one processor to another. When processes/processor = 1 no processes are preempted. In this case the difference between the non-concurrent algorithms is simply the bus contention and the fixed overhead because we are not modelling page faults. All degrade comparably, although DCAS/A&F suffers from bus-contention on the count of active threads. The Valois algorithm using CAS exploits concurrency as the number of processors increase but the overhead is large relative to the simpler algorithms. The bus and memory contention are so much greater that the

⁶It was necessary to derive our own version of the algorithm, as the algorithm presented in [23] is not strictly correct. This is the natural result of the complicated contortions necessary when using only CAS. The DCAS algorithm is relatively straightforward.

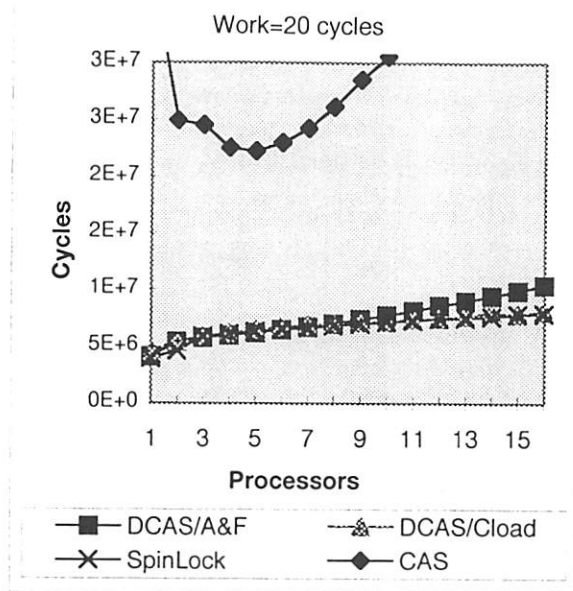


Figure 5: Performance of several synchronization algorithms with local work = 20 and the number of processes per processor = 1

concurrency does not gain enough to offset the loss due to overhead. Further, synchronization contention causes the deletion of auxiliary nodes to fail, so the number of nodes traversed increases with a larger number of processes⁷. Our DCAS algorithm performs substantially better than CAS, even with concurrency.

Figure 6 displays the results from reducing the rate of access and interleaving list accesses in parallel with the local work. Insertion/delete pairs appear to take 400 cycles with no cache interference so adding 2000 cycles of “local work” lets even the non-concurrent algorithms use about 4 or 5 processors concurrently to do useful work in parallel. Beyond that number of processors, the accesses to the list are serialized, and completion time is dominated by the time to do 10,000 insertion/deletion pairs. DCAS with either form of contention control performs comparably to spin-locks in the case of no delays and performance is significantly better than the CAS-only algorithm.

Figure 7 shows the results when 3 processes run on each processor. In this scenario, processes can be preempted — possibly while holding a lock. As is expected, spin-locks are non-competitive once delays are introduced. In

⁷The Valois simulation in Michael and Scott [18] reports better asymptotic behavior than we do. The difference appears because the authors are only simulating a FIFO queue. In the FIFO queue algorithm — where insertion always occurs at the tail and deletion at the head — auxiliary nodes are not traversed in general and thus don’t affect completion time. In fully general lists auxiliary nodes increase the execution time and memory traffic.

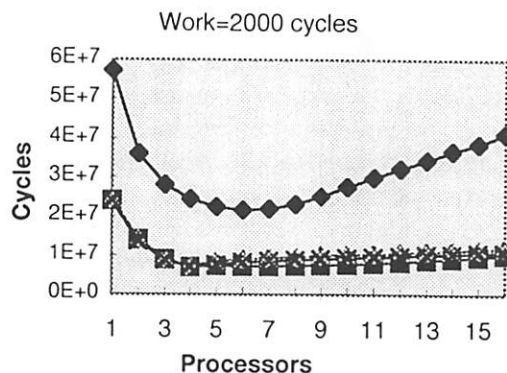


Figure 6: Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 1

contrast, the non-blocking algorithms are only slightly affected by the preemption. The completion time of CAS is mostly unaffected, however the variance (not shown in the figures) increases due to reference counts held by preempted processes delaying the deletion of nodes — when a process resumes after a delay, it can spend time releasing hundreds of nodes to the free list. These results also indicate how hardware advisory locking performs compared to operating system support in the style of Allemany and Felten. In the normal case, the lockholder experiences no delays and the waiters are notified immediately when the advisory lock is released. However, when a process is preempted, the waiters are *not* notified. When the waiter has backed off beyond a certain maximum threshold, it uses a normal Load rather than a Cload and no longer waits for the lock-holder. With a large number of processes, the occasional occurrence of this (bounded) delay enables DCAS/A&F to outperform the cache-based advisory locking. However, the expected behavior of the Cache Kernel is for the waiters to be on the same processor as the lock-holder (either signal handlers or local context switch). In this case, the advisory lock does not prevent the waiter from making progress. Therefore, there is no advantage to the operating system notification and the lower overhead of advisory locking makes it preferable.

Overall, DCAS performs comparably to, or better than, spin locks and CAS algorithms. Moreover, the code is considerably simpler than the CAS algorithm of Valois.

In these simulations, the number of processors accessing a single data structure is far higher than would occur under real loads and the rate of access to the shared data structure is far higher than one would expect on a real system. As previously noted, contention levels such as these are indicative of a poorly designed system and would

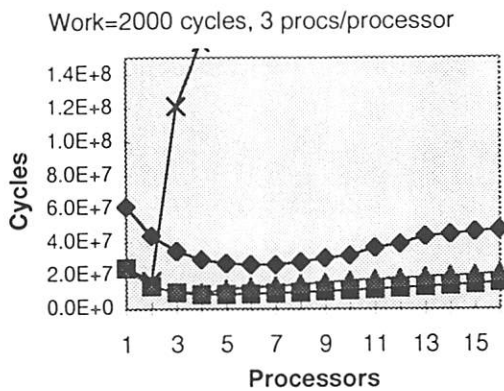


Figure 7: Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 3

have caused us to redesign this data structure. However, they do indicate that our techniques handle stress well.

7.3 Overall System Performance

We do not have the ideal measurements to show the benefit of non-blocking synchronization for overall system performance. However, in other work [24], system performance has been shown to benefit considerably from the ability to execute code in signal handlers as exploited extensively by the Cache Kernel object-oriented remote procedure call system. This system allows restricted procedures, namely those that do not block, to be executed directly as part of the signal handler invocation that handles a new call. With this optimization, many performance-critical RPCs can be invoked directly in the signal handler without the overhead of allocating and dispatching a separate thread to execute the RPC. Our measurements, reported in the cited paper, indicate a significant savings from this optimization, particularly for short-execution calls that are common to operating system services and simulations.

8 Related Work

Previous work has explored lock-free operating systems implementations, general techniques for wait-free concurrent data structures, and hardware and operating system support for non-blocking synchronization.

8.1 Lock-Free Operating Systems

Massalin and Pu [17] describe the lock-free (non-blocking) implementation of the Synthesis V.1 multiprocessor kernel, using just CAS and DCAS, the same as our work. Their work supports our contention that DCAS is sufficient for the practical implementation of large

systems using non-blocking synchronization. However, their work focused on using a small number of wait-free and lock-free data structures inside their operating system kernel. One reason their work has not been further emulated is their exploitation of application-specific optimizations to implement data structures. One example is their implementation of linked list with insertion and deletion from the middle of the list: it is efficient only because the usage within the Synthesis kernel is highly constrained and a single bit suffices where a reference count is normally needed. In contrast, our implementation of linked lists is general, and is usable by arbitrary application code.

8.2 Methodologies for Implementing Concurrent Data Objects

Herlihy [14] presents a methodology for converting sequential implementations of data structures into wait-free concurrent implementations. The goal is to provide a specification and transformation that is provably correct and can be applied automatically to sequential code. It converts a sequential implementation of any data structure into a wait-free, concurrent one, just using CAS (or, slightly more efficiently [14] using *load-linked* and *store-conditional*). However, this method involves copying the entire data structure, modifying the copy, and then atomically replacing the old copy with the new copy using CAS, and retrying the entire copy and modifying if there is a conflict. Performance can be improved using other, more ad-hoc, techniques [14], but these techniques tend to add hard-to-catch subtle synchronization problems and are still expensive. Overall, we regard this approach as impractically expensive because of the copy overhead.

In contrast, our contribution is a set of general techniques that the programmer incorporates in the software design and implementation, allowing the software to be used in both sequential and parallel execution with no modification and with acceptable performance.

Barnes [3], Turek [22], and Valois [23] provide techniques for increasing the concurrency with some non-blocking synchronization. However, the cost of concurrent updates appears to outweigh the actual benefit, because the low rates of contention in our system. Studies such as [21], which also reported a low level of contention on kernel data structures, suggest that this phenomenon might be more widely true than just in the Cache Kernel.

8.3 Hardware Support

Most processors provide at most single Compare-and-Swap (CAS) functionality to support non-blocking synchronization. Herlihy's general methodology [13] shows that that single CAS is adequate in theory but appears too inefficient in practice.

A few processors such as the Motorola 68040 provide a multi-word atomic instruction but that functionality is rare and is not present in any RISC processor to our knowledge. The RISC-like extension that we propose in Section 6 suggests that it is feasible to support in modern processors. The CISC approach does not appear viable with most current and future processors and seems likely to die out with the current processors that support it.

Transactional Memory [12] provides hardware support for multiple-address atomic memory operations. It is more general than DCAS but comes at a correspondingly higher cost. The proposed hardware implementation requires six new instructions, a second set of caches in the processor, twice the storage for cache lines actively involved in a transaction, and a more complicated “commit” protocol. Double LL/SC appears to be a more practical solution because DCAS functionality is sufficient and significantly simpler to implement.

Oklahoma Update [20] provides an alternate implementation of multiple-address atomic memory operations. Rather than duplicating entire cache lines involved in transactions (as Transactional Memory does), Oklahoma Update requires only a reservation register per word used in their version of Load Linked. This register contains flags plus two words (and optionally two more). This contrasts with our implementation which requires a “link address retained” register per synchronized word and a single cache-line buffer for the delayed SCP. Our design can also work with a word register instead of an entire cache line to buffer the SCP. However, this approach adds complexity to the chip’s logic, slows down the SC and increases the time the cache is locked so the savings are questionable. The Oklahoma Update attempts to implement some features in hardware (e.g. exponential backoff) which are better done in software, and which needlessly increase the complexity and size of the chip. Also, buffering of certain requests that come in during the “pre-commit” phase can cause two processors with non-interfering reservation sets to delay each other⁸.

These different designs arise because of different assumptions regarding the number of memory locations that should be atomically updatable at one time. Transactional Memory paper conjectures between 10 and 100 and Oklahoma Update places the knee at 3 or 4. In general, more locations are better and more powerful. However, our implementation at 2 (DCAS) is by far the simplest extension to existing processor designs. A key contribution of our work is experience that indicates that DCAS is sufficient for practical performance, making the extra

hardware complexity of the other schemes unnecessary.

8.4 Operating System Support

Allemany and Felten [1] reduce useless concurrency with OS support to provide the same functionality that we support in hardware using cache-based advisory locking. The method is a variation on the technique of Bershad discussed in Section 6.1. They propose incrementing a counter of active threads on entrance to a critical section, and decrementing on exit. The OS decrements the counter while an active thread is switched out. Processes must wait until the count of active threads is below some threshold (1, in our case) before being allowed to proceed. Delayed processes do not excessively delay other processes because the count is decremented by the OS.

These techniques appear valuable for systems without hardware support for advisory locking and in fact their approach works better than ours under high contention. However, hardware advisory locking and conditional load are more resilient to processor failure and have lower overhead in the low-contention case. As with hardware versus software DCAS, the hardware implementation is simple and fast; further measurements are required to determine if it is compellingly so.

In other work, Israeli and Rappaport [15] implement n -way atomic Compare and Swap and n -way LL/SC for P processors out of single CAS. However, this approach is primarily of theoretical interest because it requires a large amount of space (at least P bits for every word in the shared memory), requires words to be P bits wide, takes $O(P)$ to execute, and only interlocks against other multi-word atomic instructions. Anderson and Moir [2] improve upon this, requiring only realistic sized words, $O(1)$ time, but still requiring a prohibitively large amount of space.

Finally, Software Transactional Memory [19] is an attempt to implement Transactional Memory in software, depending only on LL/SC. Unfortunately, their implementation will not work correctly on existing implementations of LL/SC because their code (AcquireOwnership) depends on the ability to interleave two outstanding LL/SC’s simultaneously, which is not supported. The LLP/SCP instructions we proposed would enable their techniques to be used to provide software transactional memory for multiple, independently chosen, words of memory. However, the space and computational overhead in their implementation is excessive for general use⁹. Moreover, the STM operations are only atomic with respect to other STM operations, and not to general reads and writes.

⁸Consider processors P_1 , P_2 and P_3 . P_1 accesses cache lines Y,Z, P_2 X,Y, and P_3 W,X (addressed in ascending alphabetical order). P_1 and P_3 should not interact. However, if P_1 holds Y and Z and P_2 holds X, then when P_2 asks P_1 for Y, P_2 stalls, and buffers P_3 ’s request for X. Thus, P_1 delays P_3 . Longer chains can be constructed.

⁹Their scheme requires twice the memory for every possibly shared location and extra overhead of at least a factor of three for reads and writes even in the case of no contention.

9 Concluding Remarks

Our experience suggests that there is a powerful synergy between non-blocking synchronization and several good structuring techniques for the design and implementation of an operating system and supporting run-time libraries. Non-blocking synchronization significantly reduces the complexity and improves the performance of software in the *signal-rich* environment implemented by the Cache Kernel and supporting class libraries. Moreover, the structuring techniques we have used to achieve our overall system design goals facilitate implementing non-blocking synchronization. The biggest problem has been inadequate performance of the non-blocking synchronization instructions.

This work makes several contributions. First, we show that careful design and implementation of operating system software for efficiency, reliability and modularity makes implementing simple, efficient non-blocking synchronization far easier. In particular, type-stable memory (TSM), contention-minimizing data structuring and minimal inconsistency window structuring are important for all these reasons. These techniques are beneficial even with blocking synchronization and yet significantly reduce the complexity and improve the performance of non-blocking synchronization. Conversely, non-blocking synchronization has significant advantages in the signal-centric design of the Cache Kernel and its associated libraries, especially with the large amount of conventional operating system functionality that is implemented at the library, rather than kernel, level.

Second, we describe a number of techniques for implementing non-blocking synchronization using TSM, version numbers and DCAS. In contrast to single CAS without TSM, these techniques are simple to write, read, and understand, and perform well. Our experience suggests that good DCAS support is sufficient for a practical non-blocking OS and run-time system implementation, and that single CAS is not sufficient. In fact, lack of efficient DCAS support in systems is a potential impediment to using our techniques.

Fortunately, our proposed hardware implementation indicates that it is feasible to implement efficient DCAS functionality in a modern processor with minimal additional complexity and full compatibility with the load-store architecture. The conditional load capability coupled to cache-based advisory locking further improves the hardware support, providing the advantages of locking in a lock-free implementation. The existence of software implementations of DCAS and contention reduction demonstrates that our approach is reasonable even on platforms lacking hardware support.

Efficiently supported DCAS would allow fully-synchronized standard libraries and operating system

software to be portable across multiprocessors and uniprocessors without extra overhead or code complication. It would allow parallel architectures to use software developed for uniprocessors, relying on the (non-blocking) synchronization required for signals to handle serialization in the parallel processing context. This would significantly reduce the software bottleneck that has slowed the deployment of parallel processing to date.

Further work is required to evaluate the merits of hardware support for DCAS versus various software alternatives, particularly for overall system performance. Further work is also required to validate our experience that DCAS is in fact adequate in practice. However, our experience to date convinces us that the non-blocking approach is an attractive and practical way to structure operating system software. Locks will become more problematic as signals are used more extensively in libraries, synchronization becomes finer grained, and as the cost of memory delays and descheduling become even higher relative to processor speed. We hope our work encourages additional efforts in this area.

10 Acknowledgements

This paper has benefited from the comments of Hugh Holbrook and Ken Duda.

References

- [1] J. Allemany and E.W.Felton, Performance issues in non-blocking synchronization on shared memory multiprocessors. *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp 125-134, August 1992.
- [2] J.H. Anderson and M. Moir, Universal Constructions for Multi-Object Operations, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 184-193, August 20-23, 1995
- [3] G. Barnes, A Method for Implementing Lock-Free Shared Data Structures *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* 1993
- [4] B.N. Bershad, Practical considerations for non-blocking concurrent objects. *Proceedings 13th IEEE International Conference on Distributed Computing Systems*, Los Alamitos CA, IEEE Computer Society Press, pp 264-273, May 25-28, 1993.
- [5] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl, "PROTEUS: A High-Performance Parallel-Architecture Simulator", Technical Report

MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

- [6] D.R. Cheriton, The V Distributed System. *Communications of the ACM*, 31(3), pp 314-333, March 1988
- [7] D.R. Cheriton and K. Duda. A Caching Model of Operating System Kernel Functionality. *Proceedings of 1st Symposium on Operation Systems Design and Implementation*, Monterey, CA, pp 179-193, Nov 14-17, 1994.
- [8] D.R. Cheriton, H. Goosen, and P. Boyle, ParaDiGM: A highly scalable shared-memory multi-computer architecture. *IEEE Computer*, 24(2), February 1991.
- [9] D.R. Cheriton and R. Kutter. Optimizing memory-based messaging for scalable shared memory multiprocessor architectures. To appear in *USENIX Computer Systems Journal* 1996. (available as Stanford Computer Science Technical Report CS-93-123, December 1993.)
- [10] D.R. Cheriton, H. Goosen, and P. Machanick, Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pp 23-31, Tokyo, April 1991.
- [11] Joseph Heinrich. MIPS R4000 User's Manual, PTR Prentice Hall, Englewood Cliffs NJ, 1993
- [12] M.P. Herlihy and J.E.B. Moss. Transactional Memory: Architectural support for lock-free data structures. *1993 20th Annual Symposium on Computer Architecture* San Diego, Calif. pp. 289-301. May 1993.
- [13] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pp 123-149, January, 1991
- [14] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects *ACM Transactions on Programming Languages and Systems*, 15(5), 745-770, November, 1993
- [15] A. Israeli and L. Rappaport, Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives, *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, pp 151-160, August 14-17, 1994
- [16] A. Israeli and L. Rappaport, Efficient wait-free implementation of a concurrent priority queue *7th Intl Workshop on Distributed Algorithms '93*, Lausanne, Switzerland, *Lecture Notes in Computer Science* 725, Springer Verlag, pp 1-17, Sept. 1993
- [17] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.
- [18] M. Michael and M. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, pp 267-276, May 1996.
- [19] N. Shavit and D. Tovitov, Software Transactional Memory, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 204-213, August 20-23, 1995
- [20] J. Stone, H. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, vol 1, no.4, pp 58-71, November, 1993
- [21] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 162-174, October 1992
- [22] J. Turek, D. Shasha and S. Prakash. Locking without blocking: Making Lock-Based Concurrent Data Structure Algorithms Non-Blocking. *Proceedings of the 1992 Principles of Database Systems* pp 212-222, 1992.
- [23] J. Valois, Lock-Free Linked Lists Using Compare-and-Swap, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 214-222, August 20-23, 1995
- [24] M. Zelesko and D. R. Cheriton, Specializing Object Oriented RPC for Functionality and Performance, *Proceedings 16th IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society Press, May 27-30, 1996.
- [25] M68000 Family Programmer's Reference Manual, Motorola, Inc. 1989

Microkernels Meet Recursive Virtual Machines

Bryan Ford Mike Hibler Jay Lepreau Patrick Tullmann
Godmar Back Stephen Clawson

*Department of Computer Science, University of Utah
Salt Lake City, UT 84112*

flux@cs.utah.edu <http://www.cs.utah.edu/projects/flux/>

Abstract

This paper describes a novel approach to providing modular and extensible operating system functionality and encapsulated environments based on a synthesis of microkernel and virtual machine concepts. We have developed a software-based *virtualizable architecture* called Fluke that allows recursive virtual machines (virtual machines running on other virtual machines) to be implemented efficiently by a microkernel running on generic hardware. A complete virtual machine interface is provided at each level; efficiency derives from needing to implement only *new* functionality at each level. This infrastructure allows common OS functionality, such as process management, demand paging, fault tolerance, and debugging support, to be provided by cleanly modularized, independent, stackable virtual machine monitors, implemented as user processes. It can also provide uncommon or unique OS features, including the above features specialized for particular applications' needs, virtual machines transparently distributed cross-node, or security monitors that allow arbitrary untrusted binaries to be executed safely. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way. Some types of virtual machine layers impose almost no overhead at all, while others impose some overhead (typically 0–35%), but only on certain classes of applications.

1 Introduction

Increasing operating system modularity and extensibility without excessively hurting performance is a topic of much ongoing research [5, 9, 18, 36, 40]. Microkernels [4, 24] attempt to decompose operating systems “horizontally” by moving traditional kernel functionality into servers running in user mode. Recursive virtual machines [23], on the other hand, allow operating systems to be decomposed

“vertically” by implementing OS functionality in stackable *virtual machine monitors*, each of which exports a virtual machine interface compatible with the machine interface on which it runs. Traditionally, virtual machines have been implemented on and export existing hardware architectures so they can support “naive” operating systems (see Figure 1). For example, the most well-known virtual machine system, VM/370 [28, 29], provides virtual memory and security between multiple concurrent virtual machines, all exporting the IBM S/370 hardware architecture. Furthermore, special *virtualizable hardware architectures* [22, 35] have been proposed, whose design goal is to allow virtual machines to be stacked much more efficiently.

This paper presents a new approach to OS extensibility which combines both microkernel and virtual machine concepts in one system. We have designed a “virtualizable architecture” that does *not* attempt to emulate an actual hardware architecture closely, but is instead designed along the lines of a traditional process model and is intended to be implemented in software by a microkernel. The microkernel runs on the “raw” hardware platform and exports our software-based virtualizable architecture (see Figure 2), which we will refer to as a *virtualizable process* or *nested process architecture* to avoid confusion with traditional hardware-based architectures. The virtual machine monitors designed to run on this software architecture, which we call *nesters*, can efficiently create additional recursive virtual machines or *nested processes* in which arbitrary applications or other nesters can run.

Although the Fluke architecture does not closely follow a traditional virtual machine architecture, it is designed to preserve certain highly useful properties of recursive virtual machines. These properties are required to different degrees by different nesters that take advantage of the model. For example, demand paging and checkpointing nesters require access to and control over the program state contained in their children, grandchildren, and so on, whereas process management and security monitoring nesters primarily rely on being able to monitor and control IPC-based communication across the *boundary* sur-

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army, under contract number DABT63-94-C-0058. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies of the U.S. Government.

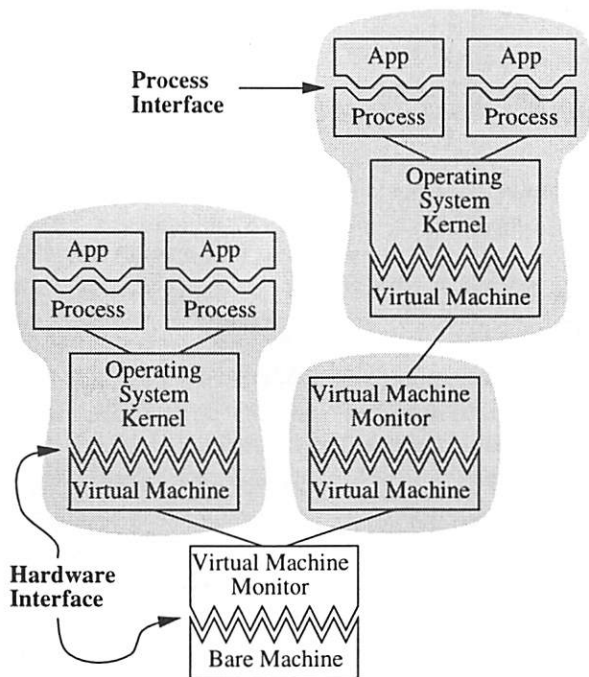


Figure 1: Traditional virtual machines based on hardware architectures. Each shaded area is a separate virtual machine, and each virtual machine exports the same architecture as the base machine's architecture.

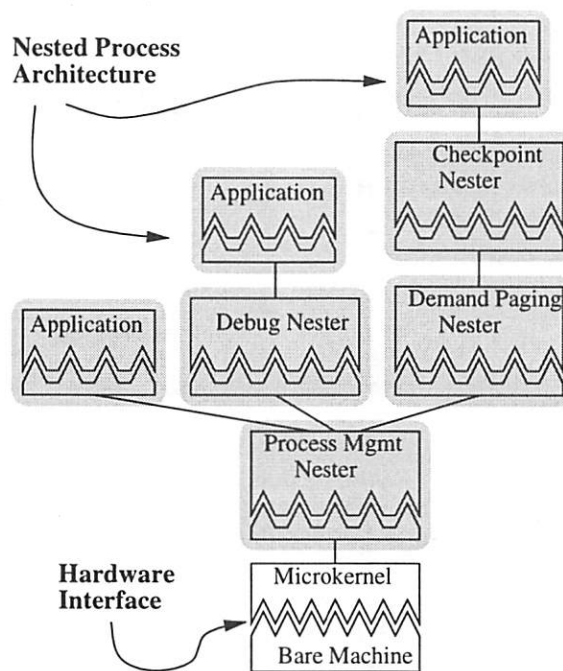


Figure 2: Virtual machines based on an extended architecture implemented by a microkernel. The interface between the microkernel and the bare machine is a traditional hardware-based machine architecture, but the common interface between all the other layers in the system is a software-based nested process architecture. Each shaded area is a separate process.

rounding the nested environment.

Our microkernel's API provides these properties efficiently in several ways. Address spaces are composed from other address spaces using hierarchical memory remapping primitives. For CPU resources, the kernel provides primitives that support hierarchical scheduling. To allow IPC-based communication to short-circuit the hierarchy safely, the kernel provides a global capability model that supports *selective* interposition on communication channels. On top of the microkernel API, well-defined IPC interfaces provide I/O and resource management functionality at a higher level than in traditional virtual machines. These higher-level interfaces are more suited to the needs of modern applications: e.g., they provide file handles instead of device I/O registers.

This nested process architecture can be used to apply existing algorithms and techniques in more flexible ways. Some examples we demonstrate in this paper include the following:

Decomposing the kernel: Some features of traditional operating systems are usually so tightly integrated into the kernel that it is difficult to eliminate them in situations in which they are not needed. A striking example is demand paging. Although it is often possible to disable it in particular situations on particular regions (e.g., using POSIX's `mlock()`), all of the paging support is still in the kernel, occupying memory and increasing system overhead. Even systems that support "external pagers," such as Mach, contain considerable paging-related code in the ker-

nel and most do not allow control over physical memory management, just backing store. Similarly, multiuser security mechanisms are not always needed, since most personal computers are dedicated to the use of a single person, and even process management and job control features may not be needed in single-application systems such as the proverbial "Internet appliance." Our system demonstrates decomposed paging and POSIX process management by implementing these traditional kernel functions as optional nesters which can be used only when needed, and only for the parts of a system for which they are desired.

Increasing the scope of existing mechanisms: There are algorithms and software packages available for common operating systems to provide features such as distributed shared memory (DSM) [10, 32], checkpointing [11], and security against untrusted applications [52]. However, these systems only cleanly support applications running in a single logical protection domain. In a nested process model, any process can create further nested subprocesses which are completely encapsulated within the parent. This design allows DSM, checkpointing, security, and other mechanisms to be applied just as easily to multiprocess applications or even complete operating environments. Our system demonstrates this flexibility by providing a checkpointer, implemented as a nester, which can be transparently applied to arbitrary domains such as a single application, a multi-process user environment containing a

process manager and multiple applications, or even the entire system.

Composing OS features: The mechanisms mentioned above are generally difficult or impossible to combine flexibly. One might be able to run an application and checkpoint it, or to run an untrusted application in a secure environment, but existing software mechanisms are insufficient to run a *checkpointed, untrusted* application without implementing a new, specialized program designed to provide both functions. A nested process architecture allows one to combine such features by layering the mechanisms, since the interface between each layer is the same. In Fluke, for example, a Unix-like environment can be built by running a process manager within a virtual memory manager, so that the process manager and all of the processes it controls are paged. Alternatively, the virtual memory manager can be run within the process manager to provide virtual memory to an individual process.

We used micro benchmarks to measure the system's performance in a variety of configurations. These measurements indicate a slowdown of about 0–35% per virtual machine layer, in contrast to conventional recursive virtual machines whose slowdown is 20%–100% [7]. Some nesters, such as the process manager, do not need to interpose on performance-critical interfaces such as memory allocation or file I/O, and hence take better advantage of the short-circuit communication facilities provided by the microkernel architecture. These nesters cause almost no slowdown at all. Other nesters, such as the memory manager and the checkpointer, must interfere more to perform their function, and therefore cause some slowdown. However, even this slowdown is fairly reasonable. Our results indicate that, at least for the applications we have tested, this combined virtual machine/microkernel model indeed provides a practical method of increasing operating system modularity, flexibility, and power.

The rest of this paper is organized as follows: In Section 2 we compare our architecture to related work. We describe the key principles upon which our work is based in Section 3, and our software-based virtualizable architecture derived from these principles in Section 4. Section 5 describes the implementation of the example applications and nesters we designed to take advantage of the nested process model. Section 6 describes the experiments and results using the example process nesters. Finally, we conclude with a short reflective summary.

2 Related Work

In this section, we first summarize virtual machine concepts and how our system incorporates them; second, we show how our design relates to conventional process models, and finally, we contrast our system with other microkernel-based systems. We describe related work concerning the details of our design later, in the appropriate

sections.

2.1 Traditional Virtual Machines

A *virtual machine simulator*, such as the Java interpreter [25], is a program that runs on one hardware architecture and implements in software a virtual machine conforming to a completely different architecture. In contrast, a *virtual machine monitor* or *hypervisor*, such as VM/370 [28, 29], is a program that creates one or more virtual machines exporting the same hardware architecture as the machine it runs on. Hypervisors are typically much more efficient than simulators because most of the instructions in the virtual machine environment can be executed at full speed on the bare hardware, and only “special” instructions such as privileged instructions and accesses to I/O registers need to be emulated in software. Since the “upper” and “lower” interfaces of a hypervisor are the same, a sufficiently complete hypervisor can even run additional copies of itself, recursively.

Virtual machines have been used for a variety of purposes including security, fault tolerance, and operating system software development [23]. In their heyday, virtual machine systems were not driven by modularity issues at all, but instead were created to make better use of scarce, expensive hardware resources. For example, organizations often needed to run several applications requiring different operating systems concurrently, and possibly test new operating systems, all on only a single mainframe. Therefore, traditional virtual machine systems used (and needed) only shallow hierarchies, implementing all required functionality in a single hypervisor. As hardware became cheap and ubiquitous, virtual machines became less common, although they remain in use in specialized contexts such as fault tolerance [7], and safe environments for untrusted applications [25].

In this paper we revive the idea of using virtual machine concepts pervasively throughout a system, but for the purpose of enhancing OS modularity, flexibility, and extensibility, rather than merely for hardware multiplexing. For these purposes, virtual machines based on hardware architectures have several drawbacks. First, most processor architectures allow “sensitive” information such as the current privilege level to leak into user-accessible registers, making it impossible for a hypervisor to recreate the underlying architecture faithfully without extensive emulation of even unprivileged instructions. Second, a hypervisor's performance worsens exponentially with stacking depth because each layer must trap and emulate all privileged instructions executed in the next higher layer (see Figure 3). Third, since hardware architectures are oblivious to the notion of stacking, all communication must be strictly parent-child; there is no way to support “short-circuit” communication between siblings, grandparents and grandchildren, etc.

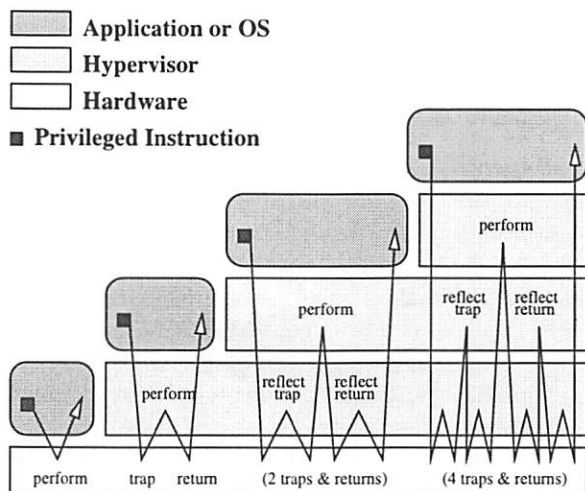


Figure 3: Exponential slowdown in traditional recursive virtual machines caused by emulation and reflection of privileged instructions, accesses to device registers, etc. For n hypervisor layers, at least $2^n - 1$ hardware trap/return sequences are required to emulate any privileged instruction executed in the top layer.

2.2 The Nested Process Model

For the above reasons, Fluke does *not* conform exactly to an existing hardware architecture. Instead, we use a software-based architecture which allows us to solve these problems by defining the architecture to avoid them. Although some existing hypervisors export a virtual architecture that differs slightly from the architecture on which they are based, they generally try to minimize such divergence, changing the architecture only enough to address “show-stopping” problems such as double paging. In contrast, Fluke does not attempt to minimize divergence, since running existing operating systems unmodified in our virtual machines is not a goal. Instead, our design goal is to maintain maximum performance in the presence of deep virtual machine layering. The resulting architecture is similar to conventional process models such as Unix’s, though with some important differences to make processes recursively virtualizable or “nestable”; hence the term “nested process architecture.”

The Cambridge CAP computer [54] implemented a similar high-level architecture in microcode supporting an arbitrarily deep process hierarchy in which parent processes virtualize memory, CPU, and trap handlers for child processes. However, the CAP computer strictly enforced the process hierarchy at all levels, and did not allow communication paths to “short-circuit” the layers as Fluke does. As noted in retrospect by the designers of the system, this weakness made it impractical for performance reasons to use more than two levels of process hierarchy (corresponding to the “supervisor” and “user” modes of other architectures), so the uses of nested processes were never actually explored or tested in this system.

System call emulation, interposition, and stacking have

been used in the past to virtualize the activity of a process by interposing special software modules between an application and the actual OS on which it is running. This form of interposition can be used, for example, to trace system calls or change the process’s view of the file system [31, 33], or to provide security against an untrusted application [52]. However, these mechanisms can only be applied easily to a single application process and generally cannot be used in combination (only one interposition module can be used on a given process). Furthermore, although file system access and other system call-based activity can be monitored and virtualized this way, it is difficult to virtualize other resources such as CPU and memory.

2.3 Other Microkernel Architectures

Our nested process model shares many of the same goals as those driving microkernel-based systems: flexibility, modularity, extensibility, and the decomposition of traditional OS features into separate, unprivileged modules. In fact, our prototype is essentially a microkernel-based system revolving around a small privileged kernel that exports a minimal set of interfaces to higher-level software. The primary difference of interest in this paper is that our microkernel is designed to support a nested process model efficiently, and the higher-level services on top of the microkernel take advantage of this model to provide a strongly structured system instead of the traditional “flat” collection of client and server processes.

Our work shares many of the extensibility goals of other current kernel-related research [5, 18, 46, 50], but takes a different, often complementary, approach. For example, the Exokernel pushes the “red line” between the privileged kernel and unprivileged code as low as possible so that application code can gain more control over the hardware. In contrast, our work is not primarily concerned with the location of this boundary, but instead with certain fundamental properties the kernel interface must have in order for processes to virtualize each other effectively.

3 Properties of Virtual Machines

Before describing the Fluke architecture in detail, we first present the underlying principles on which it is based. All of these principles derive from a single goal: to preserve the useful properties of a “pure” recursive virtual machine system without incurring the same exponential performance cost. For our purposes, a “pure” virtual machine system is one in which each layer *completely* simulates the environment of the next higher layer, including all instructions, memory accesses, and device I/O [49]. Our work hinges on two primary properties of such systems: state encapsulation and border control. These properties are described briefly in the following sections, and a concrete analysis of how our architecture satisfies these properties is presented later in Section 4.

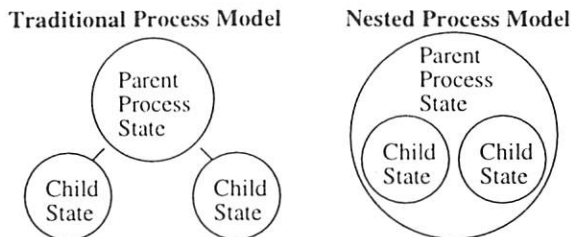


Figure 4: Process state in traditional versus nested processes. In traditional systems, the program state (code, data, heap, stack, threads, etc.) of a child process exists independently of the parent process. In a nested process architecture, the child's state is logically a part of the parent's even though the two run in separate address spaces.

3.1 State Encapsulation

The property of *state encapsulation* is the ability to encapsulate or “nest” one process inside another so that the entire state of a child process and all its descendants is logically a subset of the state of the parent process. This property allows the parent process to treat an entire child process hierarchy as merely a chunk of data that can be moved, copied, deleted, etc., while remaining oblivious to the implementation details of the child. Of primary interest are the following three specific aspects of this property: hierarchical resource management, state visibility, and reference relativity.

3.1.1 Hierarchical Resource Management

In popular systems such as Unix, Windows NT [45], and Mach [1], each process in the system exists independently of all other processes. A child process can outlive the parent process that created it, and retains all of its resources after its parent is destroyed (see Figure 4). By contrast, in a virtual machine system, a virtual machine cannot exist independently of the simulator or hypervisor that created it because the virtual machine's entire state (including the state of any sub-virtual machines it may contain) is merely part of the simulator's program variables. Some operating systems, such as L3 [39], directly support a hierarchical virtual machine-like process model in which a parent process can destroy a child process and be assured that all of the child's descendants will also go away and their resources will be freed. Other systems support a hierarchical *resource pool* abstraction from which the resources used to create processes are allocated; destroying a pool destroys any sub-pools created from it as well as all processes created from these pools. For example, KeyKOS *space banks* [6, 27] serve this purpose, as do *ledgers* in OSF's MK++ [43]. This hierarchical resource management property is a critical prerequisite for many of the applications described in this paper; without it, a parent process cannot even identify all of the state representing a child process subtree, let alone manage it coherently. Note that a traditional “flat” process model can easily be emulated in a system supporting the hierarchical model, as described in Sec-

tion 5.4, but it is difficult or impossible to emulate a hierarchical model given only a flat collection of processes.

3.1.2 State Visibility

The second aspect of the state encapsulation property is *state visibility*: the ability of a parent process to “get at” the state of a child process subtree rather than merely having control over its lifetime. State visibility is needed by any application that manages child process state, such as demand paging, debugging, checkpointing, process migration, replication, and DSM. Pure virtual machine simulators obviously satisfy this property since all of the state of a virtual machine is merely part of the simulator's variables. Same-architecture hypervisors that “delegate” to the base processor the job of executing unprivileged instructions depend on the processor to reveal the child's full register state through trap frames or shadow registers.

With notable exceptions such as the Cache Kernel [14] and Amoeba [47], most operating systems are not as good about making a child process's state visible to its parent. For example, although most kernels at least allow a process to manipulate a child's user-mode register state (e.g., Unix's `ptrace()` facility), other important state is often unavailable, such as the implicit kernel state representing outstanding long-running system calls the child may be engaged in. While these facilities are sufficient for many applications such as debugging, other applications such as process migration and checkpointing require access to *all* of the child's state.

3.1.3 Relativity of References

While full state visibility is a necessary condition to move or copy a child process, it is also necessary that its references are relative to its own scope and are not absolute references into larger scopes such as the machine or the world. A “reference” for this purpose is any piece of state in the child process that refers to some other piece of state or “object.” For example, virtual addresses, unique identifiers, and file descriptors are different forms of references. An *internal reference* is a reference in the child to some other object in the same process; an *external reference* is a reference to an object in an outer scope (e.g., in an ancestor or sibling process). For example, in Figure 5, references 1 and 2 are internal references with respect to process A, while reference 3 and 4 are external references.

In a traditional virtual machine system, external references in a virtual machine are effectively just the addresses of I/O ports or memory-mapped devices simulated by the hypervisor, and internal references are simply pointers within that machine's address space. In both cases, the representation of these references is relative to the virtual machine itself; the contents of the virtual machine can be moved at any time to a different context and all of the references it contains will remain valid.

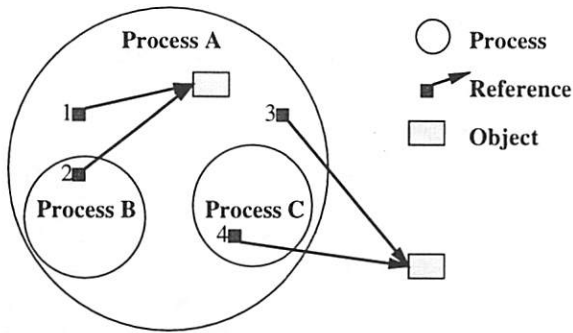


Figure 5: References into, out of, and within nested processes.

This relativity property is the exception rather than the rule in operating systems. Absolute pathnames in Unix serve as a simple example: if a process containing absolute pathnames is migrated from one machine to another, then the new machine must have *exactly* the same file system organization (at least in the parts accessed by the migrated program) or the external references from the program to the file system will become invalid.

On a system such as L3 in which process-internal objects such as threads are addressed using global unique identifiers, even *internal* references in a migrated process will become invalid unless exactly the same global identifiers can be obtained on the target machine. Furthermore, there can be no hope of cloning or replicating a process within a single global identifier namespace, since the global identifiers representing all the internal objects can only refer to one object within that scope. Implementing `fork()` in a single-address-space system [12] is a well-known instance of this problem.

3.2 Border Control

Another primary property of virtual machines is the ability of a hypervisor to monitor and control all communication across the border surrounding a virtual machine without impeding communication within the virtual machine. Barring shared memory channels, the only way for virtual machines to communicate with each other is to trap into the hypervisor, which can handle the request however it chooses. The Clans & Chiefs mechanism in L3 [38] provides border control directly in a microkernel environment. In most microkernels that use a capability model, such as Mach [1] and KeyKOS [6, 27], border control can be achieved through interposition.

Whereas the state encapsulation property allows the parent to control state *inside* the boundary, border control allows the parent to control the child's view of the world *outside* the boundary. As with state encapsulation, the extent to which border control is needed depends on the application in question. For example, Unix's ability to redirect the console output of an entire 'make' run to a file, including the output of all the subprocesses it forks off, reflects the

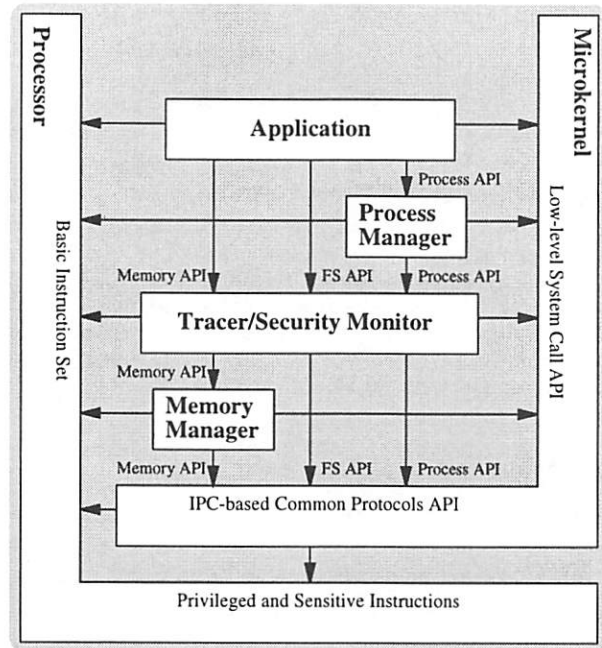


Figure 6: Illustration of the three components of the nested process architecture presented to each layer. The basic instruction set (left) used by all layers is implemented by the underlying processor and never needs to be emulated. The low-level system call API (right) is similarly implemented by the microkernel for all layers, whereas the Common Protocols API (bottom) is initially implemented by the microkernel but may subsequently be interposed on or reimplemented by higher-level layers.

limited border control provided by the capability-like file descriptor model. However, other applications, such as security monitors which allow untrusted applications to be run safely, require *complete* border control.

4 Nested Process Architecture

This section describes the nested process architecture used in our system, known as Fluke ("Flux μ -kernel Environment"). The Fluke architecture is the common interface between each of the stackable layers in the system and consists of three components: the basic computational instruction set, the low-level system call API, and the IPC-based *Common Protocols*. As Figure 6 illustrates, the key distinction between these components is that the underlying processor provides the one and only implementation of the basic instruction set used throughout the system, and the microkernel provides the only implementation of the system call API, but the Common Protocols API is independently implemented or interposed on, at least in part, at each layer. The following sections describe the three components of the nested process architecture in detail and explain how this design efficiently supports stacking while preserving all of the critical properties described in the previous section.


```

fluke_type.create(objp)
    Create a new object of type type at virtual address objp.

fluke_type.destroy(objp)
    Destroy the object at virtual address objp.

fluke_type.move(objp, new_objp)
    Move the object from virtual address objp to new_objp.

fluke_type.reference(objp, ref_objp)
    Associate the reference object ref_objp with the object objp.

fluke_type.get.state(objp, statep, refp, ...)
    Return the state of objp. State includes simple data passed back in
    the type-specific structure statep and zero or more capabilities re-
    turned in reference objects.

fluke_type.set.state(objp, statep, refp, ...)
    Loads new state into objp. State arguments are identical to those of
    the get.state call.

```

Figure 7: Example of the Fluke low-level ("system call") API. These are the operations that are common to most object types. Each object type has additional type-specific operations; e.g., IPC operations for ports and lock/unlock operations for mutexes.

4.1 Basic Computational Instruction Set

The lowest level of our nested process architecture is a well-defined subset of the x86 architecture, allowing all processes to run with no instruction emulation on x86 machines. However, other processor architectures could be used just as easily, including purely emulation-based instruction sets such as Omniware [2] or Java bytecode [25]. The only restriction is that application processes must only use the subset of the instruction set that satisfies the properties described in Section 3; otherwise they may not function properly in virtualized environments. The instructions compilers produce generally satisfy these requirements automatically, because they refer only to per-thread register state or per-process memory. However, most architectures have a few "sensitive" yet unprivileged instructions that reveal global information and therefore cannot be virtualized. For example, the x86's CPUID instruction identifies the processor on which it is executed, making it impossible to safely migrate applications that rely on this instruction.

4.2 Low-level API

The second part of our nested process architecture is a set of low-level microkernel objects and system calls. These primitives are always implemented by the microkernel directly; therefore it is critical that they be designed to support all of the virtualization properties in Section 3. The reward for carefully designing the low-level API in this way is that it is never necessary for a parent process to interpose on the kernel calls made by child processes; instead, it is always sufficient for the parent process merely to retain control over the resources to which the system calls refer. Figure 7 shows a sample of the Fluke microkernel API [20].

4.2.1 Address Spaces

The Fluke kernel supports an arbitrary number of *address spaces*, much like stripped-down Unix processes or Mach tasks; multiple threads can run in each address space. Unlike most operating systems, address spaces in Fluke are defined relative to other spaces. The low-level API does not include system calls to allocate or free memory; it only provides primitives to remap a given virtual address range from one space into another space, possibly at a different virtual address range or with reduced permissions.

For example, a parent process can use this mechanism to "donate" access to some of its memory pages to a nested child process, for that child to use as its own private storage. This child can in turn donate some of this memory to its children, and so on, forming a hierarchy of memory remappings. The parent can change or revoke at any time the permissions the child has to this memory. If a thread in a descendant process attempts to access addresses in its space whose permissions have been revoked, the thread will take a page fault, which the kernel will deliver as an IPC message to a thread in the parent.

This memory remapping mechanism is similar to that of L4 [40], Grasshopper [41], and the "f-maps" in virtual machine systems [22, 23]. Our architecture uses this mechanism to provide the state containment and state visibility properties described in Section 3. Since a child process can only access memory given to it by its parent, the parent can locate and identify all the memory comprising the state of a child (including its descendants) simply by keeping track of which pages of memory it grants to the child. A parent can control the child's use of this memory by manipulating the mappings from its space into the child's.

4.2.2 Kernel Objects

All low-level API calls are operations on a few types of primitive *kernel objects*, such as address spaces and threads. All active kernel objects are logically associated with, or "attached to," a small chunk of physical memory; this is reminiscent of tagged processor architectures such as System 38 [37] and the Intel i960XA [30]. A process can invoke kernel operations on kernel objects residing on pages mapped into its address space by specifying the virtual address of the object as a parameter to the appropriate system call. Since kernel objects are identified by local virtual addresses, this design satisfies the relativity property for user-mode references to kernel objects. In addition, there are system calls that a process can use to determine the location of all kernel objects within a given range of its own virtual address space, as well as system calls to examine and modify the state of these objects [51].

This kernel object design, coupled with the address space remapping mechanism described above, provides the state containment and state visibility properties for kernel object state as well as "plain data," allowing parent pro-

cesses to identify and gain access to *all* of the vital state comprising a child process. This mechanism is similar to the kernel object caching mechanism in the Cache Kernel [14], except that our mechanism does not impose ordering restrictions on manipulating kernel objects. For example, to take a checkpoint, a checkpointer temporarily revokes its child's access to the memory to be checkpointed, makes a system call to locate the kernel objects within this memory, makes additional kernel calls to extract the vital state from these objects (which includes the state of the child's threads, subprocesses, etc.), saves all the state of both the plain memory and the kernel objects, and finally resumes the child process. This is a very simplified description; the details of the procedure are presented later in Section 5.6. However, this is the fundamental mechanism by which not only checkpointing but other similar applications such as process migration and replication can be implemented transparently in our system.

4.2.3 Thread Objects

Multithreaded processes are directly supported in the low-level API through *thread objects*, which can be created and manipulated just like any other kernel object. Once a new thread object has been created and set up properly, it becomes an active, independent flow of control supervised directly by the kernel, executing instructions in its associated address space.

Unlike in most systems, Fluke threads provide full state visibility: a parent process can stop and examine the state of the threads in a child process at any time, and be assured of promptly receiving *all* relevant child state; i.e., everything necessary to transplant the child nondestructively. One implication of this property, which affects the low-level API pervasively, is that all states in which a thread may have to wait for events caused by other processes must be explicitly representable in the saved-state structure the kernel provides to a process examining the thread.

For example, our low-level API provides a "send request and await reply" IPC primitive as a single system call. Since the server may take arbitrarily long to service the request, and a parent process may try to snapshot the thread's state while the thread is waiting, the kernel must correctly reveal this intermediate state to the parent. In our system the kernel reveals this state by modifying the client thread's registers so that it appears to the parent that the client was calling the low-level API entrypoint that only waits for a reply without sending a message. This technique of adjusting user-level registers to reflect kernel state transitions is used throughout Fluke to provide full kernel state exportability without reducing efficiency by breaking up all system calls into minimal atomic units. The Mach IPC system [16] uses a similar technique, except that user code must explicitly check for interruption after every IPC operation and restart it manually if necessary. Also, various other Mach system calls are not cleanly interruptible at all and therefore do not

support full state visibility.

4.2.4 Capabilities

All references between low-level objects in Fluke are represented as kernel-mediated *capabilities* [37]. Each primitive object type contains a number of "capability slots." For example, each thread object contains an address space slot, which refers to the address space in which the thread is to run. Processes can store and manipulate individual capabilities using *reference objects*, which are kernel objects that hold a single capability of any type. System calls are provided to produce capabilities pointing to existing kernel objects, copy capabilities between reference objects and the special-purpose slots in other kernel objects, pass capabilities through IPC, and compare them against each other. A process only refers to a capability by the virtual address of the kernel object in which it is contained; it never has direct access to the capability itself.

Capabilities in our API provide the relativity property (Section 3.1.3) for cross-address-space references, such as references within a child process to objects implemented by its parent. Since only the kernel can access the actual contents of a capability, capabilities can be passed freely between arbitrary processes in our system, allowing communication to short-circuit the process hierarchy when appropriate. This contrasts with CAP [54], where capabilities are used only for communication within a process and all interprocess communication is strictly parent-child and based on traps. Kernel-mediated capabilities satisfy the relativity property because even though the contents of a capability are absolute in the view of the kernel (i.e., it is typically just a pointer in the kernel's address space), from the view of any user-level process, a capability appears relative since the process has no access to the actual pointer stored in the capability. A parent process can locate all of the capabilities contained in a child process, discover to which objects they point, and transparently make substitutions of these capabilities, analogous to "pointer swizzling" done in the persistence and language domains.

Capabilities also provide border control (Section 3.2). Since a parent process determines what capabilities it passes to its child, it can interpose on any of these capabilities, as well as on capabilities subsequently passed into or out of the child through communication channels on which the parent has interposed. This way, the parent can completely monitor and control all communication into or out of a process by intercepting IPC messages. However, the capability model also allows the parent to interpose *selectively* on only a subset of the capabilities provided to a child. For example, a nester that interposes on file system-related operations does not also need to intercept messages to other external services. This contrasts with L3's Clans & Chiefs model [38], where interposition is "all or nothing": if a parent wants to intercept any communication transparently, it must intercept *all* communication.

4.2.5 Scheduling

The final type of resource with which the Fluke kernel deals directly is CPU time. As with memory and communication, the kernel provides only minimal, completely relative scheduling facilities. Threads can act as schedulers for other threads, donating their CPU time to those threads according to some high-level scheduling policy; those threads can then further subdivide CPU time among still other threads, etc., forming a *scheduling hierarchy*. The scheduling hierarchy usually corresponds to the nested process hierarchy, but is not required to do so.

Our scheduling model, which has been prototyped and tested in user space but not yet as part of the Fluke kernel, is detailed in a companion paper [21]. However, only its relative, hierarchical nature is important to the nested process architecture. Other hierarchical schedulers should also work, such as the meter system in KeyKOS [27], lottery/stride scheduling [53], and SFQ scheduling [26].

4.3 High-level Protocols

While our low-level IPC mechanism provides primitive capability-based communication channels, a higher level protocol defines the conventions for communication over these channels. The key distinction between the high-level protocols and the low-level system call API is that *any* layer in the system can provide implementations of the objects defined by these interfaces, whereas only the microkernel can implement the low-level API. High-level objects automatically support the important properties in Section 3 because they are based on a capability model which allows interposition; the disadvantage is that interposition has a significant cost. We have designed our system around this trade-off by placing simple, performance-critical activities in the low-level API while leaving services that are invoked less often to be implemented by high-level objects.

Our high-level interfaces, defined in CORBA IDL, are known as the *Common Protocols* because they are common to each layer in the nesting hierarchy. The Fluke Common Protocols, modeled on existing multiserver systems [8, 33], are designed to support a POSIX-compatible environment, though many of the interfaces are generic enough to support other APIs such as Win32. Table 1 shows the currently defined interfaces and example methods for each.

The most basic interface, the Parent interface, is used for direct parent/child communication, and effectively acts as a first-level “name service” interface through which the child requests access to other services. This is the only interface that *all* nesters interpose on; nesters selectively interpose on other interfaces only as necessary to perform their function. The cost of interposition on the Parent interface is minimal because the child usually makes only a few requests on this interface, during its initialization phase, to find other interfaces of interest. The Parent interface currently provides methods to obtain initial file descrip-

Parent::	Basic parent-child interface
get_process	Get the Process Management interface
get_mem_pool	Get the memory pool for this process
Process::	POSIX process management interface
create_child	Create a new POSIX child process
exec	Execute a program
MemPool::	Memory Management interface
create_var_segment	Create a growable memory segment
create_sub	Create a sub-pool of a pool
FileSystem::	File system interface
open	Open a file
mkdir	Create an empty directory
FileDescription::	Open file and segment interface
read	Read data from a file
map	Map a file or memory segment

Table 1: Fluke Common Protocols interfaces and example methods. The top-level Parent interface acts as a name service for obtaining references to process management, memory management and other nesters.

tors (e.g., `stdin`, `stdout`, `stderr`), to find file system, memory, and process managers, and to exit. The relevant details of these interfaces will be presented in the next section as we describe specific applications that use and interpose on them.

5 System Implementation

In the following sections we overview the kernel, support libraries, and example nesters we have implemented to demonstrate the nested process model. These user-level applications take advantage of the model to provide traditional OS features, namely POSIX process management, demand paging, checkpointing, debugging, and tracing, in a more flexible and decomposed way. Table 2 lists the example nesters and the Common Protocols interfaces they interpose on.

5.1 The Microkernel

To provide the initial, base-level implementation of our nested process architecture, we developed a new microkernel that runs directly on “raw” x86 PC hardware. The initial implementation was designed primarily with portability, readability, and flexibility in mind rather than maximum performance; it is written mostly in machine-independent C and relies heavily on generic facilities provided by the Flux OS Toolkit [19]. The prototype supports kernel pre-emptibility, full multiprocessor locking, and can be configured to run under either an interrupt or process model. The kernel currently contains built-in console, serial, clock, disk, and network drivers, although it is designed to support out-of-kernel drivers in the future.

Besides implementing the low-level API used by all processes, the microkernel also implements a first-level Common Protocols interface defining the environment presented to the first process loaded (the “root” process). This

Nester	Parent	MemPool	Process	FileSys	File
Debug/Trace	✓	✓	✓	✓	✓
Process	✓		✓		
Memory	✓	✓			✓
Checkpoint	✓	✓	✓	✓	✓

Table 2: Fluke nesters and the interfaces they interpose on. FileSys is the file system interface; File is the file and memory segment interface.

initial interface is sufficient to allow various simple applications as well as nesters to be run directly on top of the microkernel: for example, it includes a basic (physical) memory allocation interface, and a minimal root file system interface which acts like a RAM disk.

Although it should be possible for a traditional monolithic kernel to implement a nested process architecture, we took a microkernel approach for the proof-of-concept, as it would be much more difficult to adapt an existing monolithic kernel because of the large source base and numerous changes that would be required. In addition, a monolithic kernel would benefit less from a nested process model because so much functionality is already hard-wired into the kernel. For example, while our checkpointer example would probably still apply, the decomposed process manager and virtual memory manager wouldn't. Because we chose the microkernel approach, our system takes the well-known "microkernel performance hit" [13] due to the additional decomposition and context switching overhead. This performance impact is made worse by the fact that our microkernel is new and entirely unoptimized.

5.2 The Libraries

In our system, traditional Unix system calls are implemented by the C library residing in the same address space as the application. These C library functions communicate with ancestor nesters and external servers as necessary to provide the required functionality. For example, each process's file descriptor table is managed by its local C library and stored in the process itself, as IPC capabilities to objects managed by file servers.¹ Our system currently provides two different C libraries: a "minimal" library for nesters and simple applications, and a full-blown BSD-based C library to support "real" Unix applications. Similar libraries could be designed to provide compatibility with other APIs and ABIs, such as Linux or Win32.

The Nesting library, linked only into nesters and not ordinary applications, provides the "parent-side" complement to the C library. For example, it contains standard functions to spawn nested subprocesses and to handle or forward a child's Common Protocols requests. Use of this library is optional. Applications can always create nested

¹ The actual files and "open file descriptions," containing seek pointers and other per-open state, are maintained by the file servers; this greatly simplifies some of the traditionally hairy Unix multiserver issues.

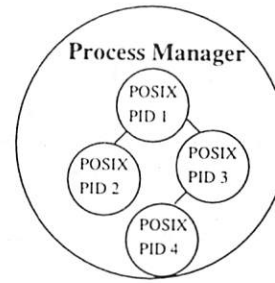


Figure 8: Process manager managing a POSIX-style process hierarchy. Each POSIX process is an immediate child of the process manager in terms of the nested process hierarchy. Process IDs and the POSIX-level parent/child relationships are implemented within the process manager.

processes manually in whatever way they desire; the library only provides a "standard" mechanism for creating children and providing Common Protocols-compatible interfaces to them.

5.3 The Debug and Trace Nesters

We have implemented, as a simple nester, a debugger that can be used to debug either ordinary applications or other nesters. The debugger works by handling exceptions that occur in the child process. When a thread in the child faults, the kernel sends an exception RPC to the debugger, which handles the fault by communicating with a remote host running GDB. The debugger restarts the thread by replying to the kernel's exception RPC. Although Mach provides a similar ability to interpose on an exception port, Mach allows a task to change its own exception port reference, potentially allowing a buggy or uncooperative task to escape the debugger's control.

We have also implemented a simple tracer, as a minor modification to the debugger, which can be used to trace the message activity of an arbitrary subprocess. The tracer starts by interposing on the application's Parent interface (as all nesters do). Then, for any request from the child that returns a new IPC capability, the tracer interposes on it. This interposition is completely generic in that the tracer needs to know nothing about the semantics of what it is interposing on; it will work for any IPC-based protocol. Although the tracer does nothing more than record messages that cross the subprocess's border, a security monitor for untrusted applications would interpose in basically the same way.

5.4 The Process Management Nester

Support for POSIX-style processes in our system is provided with the help of a *process manager*, which is the only nester in our prototype system that manages multiple immediate subprocesses at once. Each POSIX process, regardless of its location in the Unix-style process hierarchy, is a direct child of the process manager in the global nested process hierarchy, as illustrated in Figure 8. This allows

a POSIX child processes to outlive its POSIX parent, as is the proper behavior for POSIX-style processes. The Process Manager assigns a process ID to each of its child processes, and allows these processes to create other "peer" POSIX processes using `fork()` and `exec()`, which the C library converts into Common Protocols requests to the process manager. The process manager also handles inter-process signals and other POSIX process-related features. As with the other nesters, the Process Manager is an optional component: applications that don't `fork()`, send signals, etc., can be run without it. Furthermore, multiple process managers can be run side-by-side or even arbitrarily "stacked" on top of each other to provide multiple independent POSIX environments on a single machine; each of these environments has its own process ID namespace and POSIX process hierarchy.

The process manager uses the Common Protocols' *MemPool* ("memory pool") interface to manage the memory resources consumed by its children and ensure that all state is properly cleaned up when a POSIX process terminates. A *MemPool* is an object from which anonymous memory segments can be allocated and mapped by any process holding a capability for the pool. The *MemPool* interface also provides methods to create and destroy sub-pools; destroying a pool destroys all memory segments allocated from it and, recursively, all sub-pools derived from it. Thus, *MemPools* provide the hierarchical resource management property (Section 3.1.1) for memory. The process manager uses the *MemPool* capability from its parent to create a sub-pool for each POSIX process it supervises. When a process terminates or is killed, the process manager simply destroys the appropriate sub-pool; this automatically frees all memory allocated by that process and any nested subprocesses that may have been created. Furthermore, since the subprocess's threads and other kernel objects are merely part of its memory, they too get destroyed automatically when the *MemPool* is destroyed.

5.5 The Virtual Memory Management Nester

We have implemented a user-level demand paged virtual memory manager which creates a nested environment whose anonymous memory is paged to a swap file. Arbitrary programs can be run in this paged environment, such as a single application, or a process manager supporting an entire paged POSIX environment similar to a traditional Unix system. The memory manager provides paged memory by interposing on the *MemPool* interface it passes to its child, re-implementing the pool and segment operations itself instead of merely creating a sub-pool and passing the sub-pool to the child as the process manager does.

All memory segments allocated from the memory manager are backed by a swap file and cached in its own address space. The "physical memory" cache used by the memory manager is a memory segment allocated from the memory pool passed in by the memory manager's parent; the

swap file is implemented by a Common Protocols file system server.

When a client invokes the map operation on a segment implemented by the memory manager, the memory manager uses Fluke kernel address space manipulation primitives to remap the appropriate portions of its own memory into the address space of the client. These mappings correspond to "resident pages" in a conventional, kernel-based virtual memory system. The kernel notifies the memory manager of access violation and non-resident exceptions in the segments it supervises, allowing it to do demand allocation of memory and lazy copy optimization.

In the prototype memory manager, the physical memory cache is divided into fixed-size (4k) pages. All allocations and I/O operations are performed on single pages; no prepaging or other clustering techniques are used. It implements a simple global FIFO page replacement algorithm and uses a single, fixed-size file as its backing store. The current implementation does not maintain "dirty" bits, so pages are always written to backing store.

5.6 The Checkpoint Nester

We have implemented a user-level checkpointer that can operate over a single application or an arbitrary environment, transparently to the target. By loading a checkpointer as the "root" nester immediately on top of the microkernel, a whole-machine checkpointed system can be created. To our knowledge this is the first checkpointer that can operate over arbitrary domains in this way.

Like the memory manager, the checkpointer interposes on the *MemPool* interface in order to maintain complete control over the state of its child processes. Since the kernel provides primitives to locate and manipulate the low-level objects within a memory region, the checkpointer effectively has direct access to all kernel object state in the child as well as to its raw data. The checkpointer currently uses a simplistic sequential checkpointing algorithm to take a checkpoint: it temporarily revokes all permissions on the memory segments it provides (which also effectively stops all threads in the child since they have no address space to execute in), saves the contents of the child's memory and kernel object state, and then re-enables the memory permissions to allow the child to continue execution. This algorithm, of course, will not scale well to large applications or distributed environments. However, more efficient checkpointers based on well-known single-process algorithms [15, 17] could be implemented in our environment in the same way, and should also work automatically over multi-process domains.

In order to checkpoint kernel objects that contain capabilities, the checkpointer discovers what object each capability points to and replaces it with a simple identifier that is unique within the saved checkpoint image. There are two classes of capabilities that the checkpointer must deal with,

corresponding to the two types of references described in Section 3.1.3. To handle capabilities representing internal references (references to other objects within the checkpointed environment), the checkpointer builds a catalog of the objects in the checkpointed environment and uses kernel primitives to look up each capability in this catalog. Capabilities representing external references (references to objects outside the checkpointed environment) will not appear in the catalog, but since any external reference owned by the child environment must have been granted to it by the checkpointer at some point, the checkpointer can recognize it and take a reasonable course of action.

For example, the capabilities representing the `stdin`, `stdout`, and `stderr` file handles are recognized by the checkpointer and, on restart, are reinitialized with the corresponding file handles in the new environment. Thus, all standard I/O file descriptors (including descriptors in nested subprocesses of the application) are transparently routed to the new environment. Similarly, when the child makes a Parent interface request for an external service such as process management, the checkpointer keeps track of the IPC capability returned, so that it can route it to the new environment. IPC capabilities that the checkpointer chose not to interpose on, such as open files other than the standard I/O handles, are replaced with null references. This has similar consequences to an NFS server going down and leaving stale file handles behind. Although our current implementation doesn't interpose on any file system accesses, it could easily recognize open calls and save file names, or even whole files, in order to provide a more consistent restart.

The checkpointer is our most comprehensive nester, taking advantage of all of the virtual machine properties described in Section 3 to provide the most complete encapsulation possible. The state encapsulation, visibility, and relativity properties allow the checkpointer to save and restore the state of the child's memory and kernel objects. Additionally, the border control provided by the capability model allows the checkpointer to interpose on whichever interfaces it needs to; our implementation interposes on only those things necessary for a minimal complete checkpoint, comparable to the functionality offered by other user-level checkpointers [42, 48].

There is a large body of work on checkpointers both in and out of the kernel. A few existing operating systems, such as KeyKOS [34] and L3 [39], have implemented checkpointing on a whole-machine basis in the kernel. Similarly, a hypervisor was recently used to provide fault tolerance (replication) on a whole-machine basis on PA-RISC machines [7]. While these features appear practical and useful in some situations, they are inflexibly tied to the machine boundary and cannot easily be used on smaller scopes, such as a process or a group of processes, or on larger scopes, such as networked clusters of machines. The nested process model allows checkpointing and other algo-

ritms to be implemented over more flexible domains.

6 Experimental Results

In order to evaluate the performance effects of nesting processes in our Fluke implementation, we used a set of micro benchmarks designed to reveal operating system performance properties that directly affect real-world applications. Our primary interest in these tests is to show the performance effect of different nesters on various types of applications; thus, we are chiefly concerned with relative slowdown due to nesting rather than the absolute performance of the system. All tests were performed on 200MHz Pentium Pro PCs with 128MB of RAM. The micro benchmarks used are:

memtest: This is a simple memory tester which allocates as much memory as it can with `sbrk` and then makes four passes over it. Each pass writes and then reads back a distinct pattern looking for bit errors. We configured the heap to 4MB for this test. Memtest stresses memory management and is representative of programs that first allocate memory and then repeatedly operate on it.

appel2: This benchmark tests a combination of basic virtual memory primitives as identified by Appel and Li [3]. This test, known as "trap+protN+unprot" in the original paper, measures the time to protect 100 pages, randomly access each page, and, in the fault handler, unprotect the faulting page. Where memtest emphasizes higher-level Common Protocols memory management, appel2 stresses the microkernel's low-level memory remapping mechanism.

forktest: This program exercises the POSIX `fork()` and `wait()` operations of the Process Manager. Forktest creates a parent-child chain four levels deep; each level performs a simple computation and then returns the result in its exit status.

readtest: This test is similar to the `lmbench` [44] `bw.file.read` test. It reads an 8MB file in 8KB chunks accessing every word of the buffer after each read. It is intended to discover the best-case file caching capability of the operating system.

matconn: The final benchmark is a computationally intensive application. It uses Warshall's algorithm to compute connectivity in adjacency matrices, and generates successively larger matrices starting at 2x2 and ending with 128x128.

6.1 Absolute Performance

To provide a baseline for further evaluation, Table 3 presents absolute times for various primitive Fluke microkernel operations. Table 4 shows absolute times for the aforementioned benchmark programs running directly on top of the microkernel with no intervening processes. For reference, we also show micro benchmark performance results for FreeBSD (version 2.1.5). Fluke performs reason-

	Time (μ s)
Null system call	2.0
Mutex lock	3.6
Mutex unlock	4.0
Context switch	7.5
Null cross-domain RPC	14.9

Table 3: Absolute performance of microkernel primitives.

ably in most tests even though it is, for the most part, an unoptimized microkernel and FreeBSD is a mature, well-optimized monolithic kernel. There is no time listed for forktest in the Fluke column since the Fluke kernel's Common Protocols server does not implement the Process Management interface.

6.2 Overhead of Interposition

Figure 9 illustrates the overhead associated with interposition. Each benchmark was first run under the "bare" environment created by the microkernel to serve as a baseline. Then the benchmarks were run under one to four levels of the previously described trace nester which interposes on all IPC channels but simply passes data along. In this experiment, readtest is the only benchmark exhibiting measurable slowdown. This is because this test's running time is dominated by data copying since each interposition on the FileDescription read method results in an extra copy as data passes through the tracing nester's address space. These large IPC data transfers are unoptimized in the prototype kernel.

6.3 Performance of Various Nester Hierarchies

Figure 10 illustrates the effect of increasing levels of nesting on the micro benchmarks. The nester stacks were chosen to be representative of "real world" situations. The no-nesters case reflects a real-time or embedded system where only minimal OS services are required. Addition of the process manager (P) and memory manager (M) shows increasing levels of functionality, providing multitasking and traditional virtual memory services. A checkpointer (C) can be included to checkpoint or migrate the entire environment on demand. No checkpoints are actually taken during these tests, but the checkpointer performs all of the interposition and state management activities required to be able to take a checkpoint at any time. The final stack shows the insertion of the tracing nester (T) where it might be used to trace Common Protocols activity.

The matconn and appel2 benchmarks are largely unaffected by nesting. The matconn benchmark is not impacted because it is largely computational. The appel2 result demonstrates the importance of the low-level kernel API directly supporting the virtualization properties described in Section 3. Since nesters do not interpose on the low-level interface, increasing the level of nesting has minimal

Test	Fluke	FreeBSD
memtest	929091 μ s	914918 μ s
appel2	5372 μ s	3620 μ s
forktest	N/A	2534 μ s
readtest	125844 μ s	153010 μ s
matconn	102917 μ s	71568 μ s

Table 4: Absolute micro benchmark results for Fluke and FreeBSD. Fluke times reflect benchmarks running directly on top of the microkernel's minimal Common Protocols server. FreeBSD times were collected in single-user mode. All times were measured using the Pentium time-stamp counter.

impact on appel2, even though the nesters may be virtualizing memory. There is a modest (1–4% per level) kernel overhead associated with these nesters if they remap memory through themselves into their children. By mapping a child's memory into its own address space, a nester creates an additional level of mapping hierarchy in the kernel which may need to be traversed when performing virtual-to-physical address translations.

The readtest benchmark is not significantly affected by most nesters since they do not interpose on the file system interface. As in the complete interposition case, there is a significant impact (34%) when the tracing nester is added.

As expected, memtest is only impacted by memory interposition. An interesting observation is that memtest is affected to a much greater degree by the memory manager than by the checkpointer even though both interpose and remap the memory used. The difference is the way in which they remap the memory allocated for a child's segment. The current memory manager provides memory to a segment using single-page sized mappings to make page-level pagein and pageout easy, while the checkpointer uses one large multi-page mapping. The prototype kernel is not yet optimized to deal with the large number of kernel data structures that result from the memory manager's behavior.

The forktest benchmark shows a pattern similar to that of memtest, except that the performance degrades more severely. The greater impact is due to the large ratio of Common Protocols calls to other activity.

6.4 Status

Besides the simple tests and nesters above, our system is just starting to run some larger applications including GNU make, gawk, sed, bash, and gcc. We expect to make a full, self-hosting public release within the year.

7 Conclusion

In this paper we have presented a novel approach to providing modular and extensible operating system functionality based on a synthesis of microkernel and virtual machine concepts. We have demonstrated the design and im-

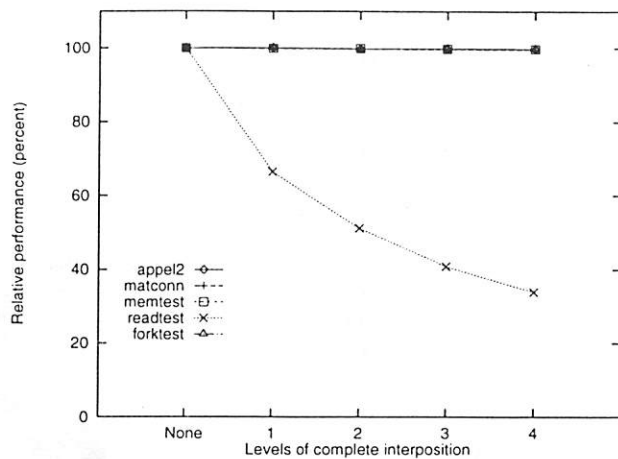


Figure 9: Worst-case overhead due to IPC interposition. Performance is measured relative to the appropriate base (no interposition) case for all tests.

plementation of a microkernel architecture that efficiently supports decomposition of traditional OS services such as process management, demand paging, fault tolerance, and debugging, into cleanly modularized, stackable layers. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way. Initial micro benchmark results are encouraging, showing slowdowns of 0–30% per layer.

Acknowledgements

For their many thoughtful and detailed comments on earlier drafts we thank our patient shepherd Marc Shapiro, the many anonymous reviewers, and Fred Schneider. Jochen Liedtke provided critique and discussion which influenced our kernel design. We thank Steve Smalley and Jeff Turner for their courage in being early guinea pigs and their critical intellectual involvement. Discussions with Jonathan Shapiro also influenced the kernel, while Michael Bushnell and Miles Bader provided key insights on the design of higher-level services. Gary Koob gave us the freedom and trust to try something this new. Finally, we are grateful to all members of the Flux project team for substantial and varied help along the way, including Chris Alfeld, John Carter, Eric Eide, Shantanu Goel, Linus Kamb, Gary Lindstrom, John McCorquodale, Roland McGrath, Bart Robinson, Sai Susarla, Kevin Van Maren, and Teresa Whiting.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.
- [2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation*, pages 127–136, May 1996.

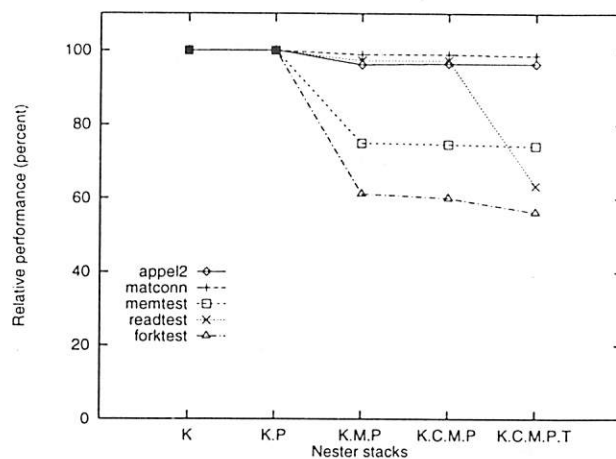


Figure 10: Slowdown due to increasing levels of process nesting. Nester stacks on the horizontal axis were chosen to be representative of actual usage. In each stack, 'K' is the kernel's Common Protocols server, 'P' is the process manager, 'M' is the memory manager, 'C' is the checkpointer, and 'T' is the tracer.

- [3] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, June 1991.
- [4] N. Batlivala, B. Gleeson, J. Hamrick, S. Lumdal, D. Price, and J. Soddy. Experience With SVR4 Over Chorus. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 223–241, Seattle, WA, Apr. 1992.
- [5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.
- [6] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.
- [7] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 1–11, Dec. 1995.
- [8] M. I. Bushnell. Towards a New Strategy of OS Design. In *GNU's Bulletin*, Cambridge, MA, Jan. 1994. Also <http://www.cs.pdx.edu/~trent/gnu/hurd-paper.html>.
- [9] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, Nov. 1994. USENIX Assoc.
- [10] J. Carter, D. Khandekar, and L. Kamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, May 1995.
- [11] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.
- [12] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. Technical Report UW-CSE-93-04-02, University of Washington Computer Science Department, Apr. 1993.
- [13] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 120–133, 1993.

- [14] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Assoc., Nov. 1994.
- [15] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory Katholieke Universiteit Leuven, Belgium, June 1993.
- [16] R. P. Draves. A Revised IPC Interface. In *Proc. of the USENIX Mach Workshop*, pages 101–121, Oct. 1990.
- [17] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, Oct. 1992.
- [18] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.
- [19] B. Ford and Flux Project Members. The Flux Operating System Toolkit. University of Utah. Postscript and HTML available under <http://www.cs.utah.edu/projects/flux/oskit/html/>, 1996.
- [20] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ -kernel Environment (draft documents). University of Utah. Postscript and HTML available under <http://www.cs.utah.edu/projects/flux/fluke/html/>, 1996.
- [21] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.
- [22] R. P. Goldberg. Architecture of Virtual Machines. In *AFIPS Conf. Proc.*, June 1973.
- [23] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, pages 34–45, June 1974.
- [24] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proc. of the Summer 1990 USENIX Conf.*, pages 87–96, Anaheim, CA, June 1990.
- [25] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as <http://java.sun.com/doc/language/environment/>.
- [26] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler For Multimedia Operations. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.
- [27] N. Hardy. The KeyKos Architecture. *Operating Systems Review*, Sept. 1985.
- [28] IBM Virtual Machine Facility /370 Planning Guide. Technical Report GC20-1801-0, IBM Corporation, 1972.
- [29] IBM Virtual Machine Facility /370: Release 2 Planning Guide. Technical Report GC20-1814-0, IBM Corporation, 1973.
- [30] Intel. *i860 Extended Architecture Programmer's Reference Manual*, 1994.
- [31] M. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 80–93, Dec. 1993.
- [32] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter USENIX Conf.*, pages 115–132, Jan. 1994.
- [33] Y. A. Khalidi and M. N. Nelson. Extensible File Systems in Spring. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 1–14, 1993.
- [34] C. Landau. The Checkpoint Mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, Sept. 1992.
- [35] H. C. Lauer and D. Wyeth. A Recursive Virtual Machine Architecture. In *ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, pages 113–116, Mar. 1973.
- [36] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 153–164, Monterey, CA, Nov. 1994. USENIX Assoc.
- [37] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.
- [38] J. Liedtke. Clans and Chiefs. In *Proc. 12. GI/ITG-Fachtagung Architektur von Rechensystemen*, 1992.
- [39] J. Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, Dec. 1993.
- [40] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.
- [41] A. Lindstrom, J. Rosenberg, and A. Dearle. The Grand Unified Theory of Address Spaces. In *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.
- [42] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. of the Winter 1992 USENIX Conf.*, 1992.
- [43] K. Loeper et al. MK++ Kernel Executive Summary. Technical report, Open Software Foundation, 1995.
- [44] L. McVoy and C. Staelin. Imbench: Portable Tools for Performance Analysis. In *Proc. of 1996 USENIX Conf.*, Jan. 1996.
- [45] Microsoft Corporation. *Win32 Programmer's Reference*, 1993. 999 pp.
- [46] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-oriented Operating System. Technical Report 94–20, University of Arizona, Dept. of Computer Science, June 1994.
- [47] S. J. Mullender. *Experiences with Distributed Systems*, chapter Process Management in Distributed Operating Systems. Lecture Notes in Computer Science no. 309. Springer-Verlag, 1987.
- [48] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. of the Winter 1995 USENIX Technical Conf.*, Jan. 1995.
- [49] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Fast and Accurate Multiprocessor Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4), 1995.
- [50] C. Small and M. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.
- [51] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level Checkpointing Through Exportable Kernel State. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct. 1996. IEEE.
- [52] D. Wagner, I. Goldberg, and R. Thomas. A Secure Environment for Untrusted Helper Applications. In *Proc. of the 6th USENIX Unix Security Symp.*, 1996.
- [53] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.
- [54] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, NY, 1979.

Making Paths Explicit in the Scout Operating System

David Mosberger and Larry L. Peterson

Department of Computer Science

University of Arizona

<http://www.cs.arizona.edu/scout/>

Abstract

This paper makes a case for *paths* as an explicit abstraction in operating system design. Paths provide a unifying infrastructure for several OS mechanisms that have been introduced in the last several years, including fbufs, integrated layer processing, packet classifiers, code specialization, and migrating threads. This paper articulates the potential advantages of a path-based OS structure, describes the specific path architecture implemented in the Scout OS, and demonstrates the advantages in a particular application domain—receiving, decoding, and displaying MPEG-compressed video.

1 Introduction

Layering is a fundamental structuring technique with a long history in system design. From early work on layered operating systems and network architectures [12, 32], to more recent advances in stackable systems [27, 15, 14, 26], layering has played a central role in managing complexity, isolating failure, and enhancing configurability. This paper describes a complementary, but equally fundamental structuring technique, which we call *paths*. Whereas layering is typically used to manage complexity, paths are applied to layered systems to improve their performance and to solve problems that require global context.

We begin by developing some intuition about paths. A path can be viewed as a logical channel through a multi-layered system over which I/O data flows, as illustrated in Figure 1. In this way, a path is analogous to a virtual circuit that cuts through the nodes of a packet-switched network. The only difference is that paths are within a single host, while virtual circuits run between hosts.¹

Also, the term “path” is well entrenched in our vocabulary. For example, we often refer to the “fast path”

¹The obvious next step is to integrate paths through the end host with circuits between hosts, but for the purpose of this paper, we focus on paths *within* a single system.

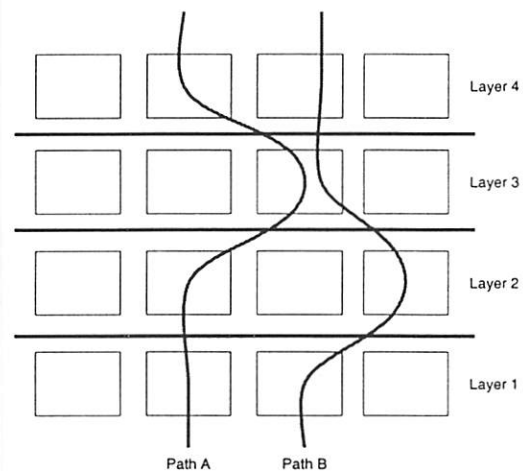


Figure 1: Two Paths Through a Layered System

through a system, implying that the most commonly executed sequence of instructions have been optimized. As another example, we sometimes talk about optimizing the “end-to-end path,” meaning we are focused on the global performance of the system (e.g., from I/O source to sink), rather than on the local performance of a single component. As a final example, we sometimes distinguish between a system’s “control path” and its “data path,” with the former being more relevant to latency and the latter more concerned with throughput.

Finally, paths can be loosely understood by considering specific OS mechanisms that have been proposed over the last few years. Consider the following examples.

- Fbufs [6] are a path-oriented buffer management mechanism designed to efficiently move data across a sequence of protection domains.² Fbufs depend on being able to identify the path through the system over which the data will flow.

²Although layering does not imply multiple protection domains, systems often impose hardware-enforced protection at layer boundaries.

- Integrated layer processing (ILP) [4, 1] is a technique for fusing the data manipulation loops of multiple protocol layers. It depends on knowing exactly what sequence of protocol modules a network packet will traverse.
- Packet classifiers [31, 20, 2, 8] distinguish among incoming network packets based on certain fields found in their headers. In a sense, a packet classifier pre-computes the path that a given message will follow.
- Specialization is sometimes used to optimize common path code sequences [24, 23]. Specialization, in turn, depends on the existence of invariants that constrain the path through the code that is likely to be executed.
- The Alpha OS allows threads to migrate across a sequence of protection domains [5]; others have defined similar mechanisms [13, 9]. Such mechanisms recognize that tasks often span multiple domains, and so account for resource usage on a path basis rather than a domain basis.

The thesis of this paper is that these mechanisms are not isolated optimizations, but rather, that they can be unified and explained by the path abstraction. In a nutshell, these mechanisms all share the following fundamental idea: they expose and exploit non-local context.

Consider a layered system like the one illustrated in Figure 1. While the advantage of layering and modularity is to hide information, there are many situations when it would be beneficial for a given layer to have access to non-local context. For example, suppose one of the modules is processing an Ethernet packet. With only local context, the module knows nothing about the packet's relative importance compared to other packets. However, if it is known that the packet is part of a particular video stream, then it is easy to determine its processing deadline, what modules need to be executed to process it, how many CPU cycles this processing will require, where its data should be placed in memory, and so on. In other words, by knowing a certain set of invariants (e.g., that the packet is part of some video stream), the module is able to access and exploit global context that is available outside any one module or layer. Abstractly then, a path is defined by these invariants and provides access to the corresponding context.

Having access to non-local context leads to two kinds of advantages: (1) improved resource allocation and scheduling decisions, and (2) improved code quality. In the former case, work is segregated early, facilitating the following benefits:

- The system can place data in a memory buffer that is already accessible to all the modules along the path. This is essentially what fbufs do. In contrast, data often has to be copied (either logically or physically) from one buffer to another at each module or layer boundary.
- The system can know that a particular path needs to be scheduled for execution in order to meet a deadline; e.g., display a video frame. This is critical to being able to offer different Qualities of Service (QoS). In contrast, not segregating work into paths means that low-priority work may need to be done to discover high-priority work that needs attention.
- If scheduling deadlines for a particular path are such that it is impossible to make use of a particular piece of work (e.g., network packet or video frame), then the system can discard unnecessary work early, that is, before executing the path. A conventional system often has to process several layers before knowing that continuing is of no value.

In the latter case—improved code quality—the system has more information available to it, making more aggressive code optimizations possible. Examples of such optimizations include the following:

- The more invariants the system knows about code to be executed, the more opportunities the system has to specialize the code path. For example, the system can do constant folding and propagation, dead-code elimination, and interprocedural register allocation.
- The more layers across which the system is able to optimize, the more opportunities there are to eliminate redundant work. For example, the more protocol layers available, the more loads and stores integrated layer processing can remove. Similarly, it is sometimes possible to merge per-layer operations. For example, instead of having each layer check for the appropriate header length, it is possible to check for the sum of all header lengths at the beginning of packet processing.

This paper makes two contributions. First, it develops an explicit path abstraction; Section 2 explores the design space for paths, and Section 3 describes an implementation of paths in the Scout operating system. Second, the paper demonstrates how having a path abstraction leads to the first set of advantages outlined above, i.e., those that have to do with improvements in resource allocation and scheduling. In particular, Section 4 describes an application that receives MPEG-compressed video over a network and then decodes and displays it. A companion paper demonstrates some of the code-related improvements attributable to paths [23].

2 Path Abstraction

While it is tempting to view paths as an optimization that can be super-imposed on an existing layered system—and it is certainly the case that many of the ideas described in this paper can be applied in this way—we take a more “first principles” approach to defining paths. Specifically, this section develops a working definition of paths in an incremental fashion. Our goal is to explore the design space for paths, and in the process, to introduce the particular architecture that we settled on.

2.1 Basic Paths

A path is a linear flow of data that starts at a source device and ends at a destination device. While the data is moved from the source to the destination, it is transformed (processed) in some path-specific manner. That is, if the input data is represented as a message m , then the output message is $g(m)$, where g is the function that represents the path-specific transformation. In addition, each path has two queues associated with it, as depicted in Figure 2, thereby decoupling the input and output devices. A path scheduler determines when a given path is executed, that is, when $g(m)$ is evaluated.

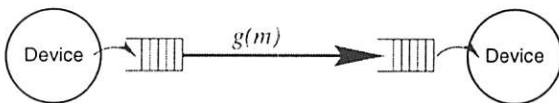


Figure 2: Simple path

While paths connect devices, there is no direct relationship between device pairs and paths: a given device pair can be connected by zero or more paths. Allowing for multiple paths to connect the same pair of devices is sensible since both $g(m)$ and the scheduling priority may vary. For example, one path may handle UDP packets, whereas another may handle TCP packets. Similarly, two paths that forward IP packets between a pair of devices may need to be scheduled differently if they provide a different quality of service. Thus, paths are dynamic entities that are created at runtime; there is no a priori limit on the number of paths that can exist in a given system.

What are the properties of basic paths? First and foremost, once a message has been enqueued on the input queue of a path, it is already known what device the (possibly transformed) output message will arrive at. For the purpose of resource management, it is also known that m belongs to the path on which it is enqueued, and all execution is performed on behalf of that path. In other words, knowledge is available *early* and *globally*.

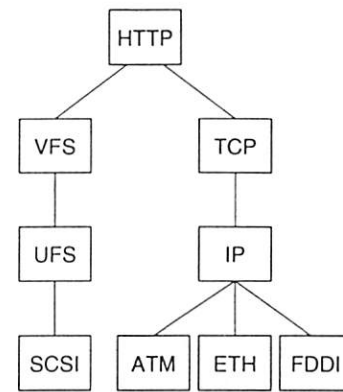


Figure 3: Example Router Graph

2.2 Creating Paths

The key problem in creating a path is how to specify $g(m)$. While it would be possible to write a specific function for each path with distinct functionality, it is more convenient to derive $g(m)$ automatically from a modularly organized system. This is because many of the paths are likely to have substantial functionality in common, such as various network, file, and windowing protocols. Generating a path from components does not preclude writing a specialized $g(m)$ for those cases that warrant the extra effort.

A *router graph* represents the modular structure of the system, where each router is a software module that implements a specific task, such as the NFS protocol or a SCSI driver. The reason we call these modules routers will become clear in a moment. As is common in a modular/layered system, individual routers provide their functionality based on the functionality of other routers, that is, it is possible to draw a dependency graph that represents the interdependence among the various routers. This also means that a router graph fully describes what kind of tasks a given system can perform. For example, Figure 3 depicts a router graph that could be used to implement a web server. Given this graph, a path that starts at the ETH router (Ethernet device) and ends at the SCSI router (disk device) would have a $g(m)$ that is the composite of the functions contributed by each router; i.e.,

$$g(m) = g_{\text{SCSI}}(g_{\text{UFS}}(\cdots(g_{\text{IP}}(g_{\text{ETH}}(m)))\cdots)).$$

However, this still begs the question of how a path is created. There are essentially two approaches: (1) the path is pre-specified externally, and (2) the path is discovered incrementally. This division corresponds to the two sources of “knowledge” that influence path creation: global and router-specific. Global knowledge is of the sort “for a web service, the following sequence of routers

need to be part of the path.” Global knowledge may also be used for optimization, for example, there may be an optimized $g(m)$ available for the web path that is preferred to an automatically derived composite function. In contrast, local knowledge is of the form “if invariant X is true for the path under construction, then the path can pass through this router” or “if invariant Y is false, then the path cannot go beyond this router.”

Using global knowledge alone to create a path would be difficult since this knowledge often requires familiarity with the internal workings of the routers that are traversed by a path. In contrast, creating a path based on router-specific knowledge alone would limit the utility of paths considerably. (Recall that most advantages of paths are due to the global knowledge they afford).

In our path architecture, paths are created in two phases. First, router-specific knowledge is used to create a maximum length path. Second, this maximum length path is transformed (optimized) using global transformation rules, each of which is defined by a $\langle \text{guard}, \text{transformation} \rangle$ pair. If the guard evaluates to TRUE, the corresponding transformation is applied, resulting in a new path. This process repeats until all guards evaluate to FALSE.

To better illustrate the difference between local and global knowledge, consider the router graph given in Figure 3. Suppose there is a path that starts at SCSI and ends at ETH. Such is the case, for example, if IP can determine that the remote host is on the same Ethernet as the local host. If this is not true, then IP can not be sure whether data will go out through ATM or FDDI, since the routing tables may change in the middle of the data transfer. Clearly, this decision is completely IP specific, that is, based on local knowledge. On the other hand, there are several global facts that hold for a web path that could be exploited, for example, data is transferred through TCP in a predominantly uni-directional manner and accessed on the disk in a strictly sequential fashion. Note that each such invariant may affect the function g_i of one or more other routers. For example, the fact that data is accessed sequentially may mean that it is best to avoid caching in the file system (UFS). Similarly, the fact that TCP is in the path may mean that the IP fragmentation code can be omitted completely.

Finally, note that our definition of a path’s semantics, which we denote as $g(m)$, should not be taken to mean that procedures constitute the fundamental building blocks of paths. It is equally legitimate to construct a path from a sequence of basic blocks, which is more in line with having the path abstraction represent the “fast path” through the system. In fact, procedures and basic blocks define two ends of a granularity spectrum. Scout implements a specific point on this spectrum, as described in Section 3.

2.3 Network View of Paths

We motivated routers as a means to automatically derive the path function $g(m)$. Alternatively, a router graph can be viewed as a set of interconnected nodes that forward messages along their links in the dependency graph. The operation of a router is to receive a message, process it, and then forward the resulting message to another router, as illustrated in Figure 4.

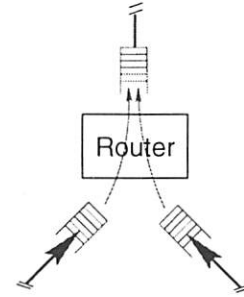


Figure 4: Message forwarding/routing

If a message is injected at some router, the trail it leaves in the router graph as it is processed and forwarded corresponds to a sequence of routing decisions. If a given trail is used very often, it may be worthwhile to explicitly encapsulate and optimize it. This is exactly what a path does: it represents a fixed sequence of routing decisions through the system’s modules. This is not unlike a virtual circuit through a network: at connection establishment time, a set of invariants that are guaranteed to hold for the duration of a connection is specified. In return, these invariants permit the customization of the path in a way that is optimized for that particular connection. In the case of a virtual circuit, the set of invariants contains the address pair of the communicating peers, but it may include other parameters such as the desired quality of service. The same kind of invariants are useful for creating paths. For example, knowing what quality-of-service a path requires helps when choosing an appropriate scheduling policy and priority.

In this context, it makes more sense to let paths connect an arbitrary pair of routers rather than insisting that a path connects a pair of devices. The latter case is ideal in the sense that it provides a maximum amount of global knowledge. However, the maximum length of a path is related to the strength of the invariants. In general, the stronger the invariants, the more routing decisions can be frozen at path-creation time, and the longer the resulting path. While it is preferable to have long paths, a general model must allow for the degenerate case where invariants are so weak that not a single routing decision can be made at path creation time. This degenerate case roughly corresponds to a traditional layered system.

2.4 Generalized Paths

As defined so far, paths are simple and highly predictable: a message arrives at the input queue, the path is scheduled for execution, and the transformed message is deposited in the output queue. While this simplicity is ideal for the purpose of optimization, it also limits the usefulness of paths. Since it is our goal to define paths in a way that moves them from a purely performance-motivated concept into an abstraction with which a complete operating system can be built, we must extend paths to make them more widely applicable, but in a way that does not destroy the properties that make the path abstraction attractive in the first place.

2.4.1 Directionality

Processing in a path is usually bi-directional: a remote-procedure call arrives over the network, results in some computation, and an answer is sent back to the caller; the arrival of a network packet triggers the sending of an acknowledgment; or a disk block is requested and arrives asynchronously. Such bi-directional paths could be handled by creating two separate paths, but it seems more natural if a path that is used to make a request is also the one that yields the response. A similar argument can be made about resource management. A more technical argument for making paths bi-directional is that often the two directions are dependent on each other. For example, when sending a network packet to a remote host, it may be desirable to include a piggy-back acknowledgment in that same packet.

Therefore, we extend the path model as follows. Each path end has a *pair* of queues—an input queue for one direction, an output queue for the other direction. The path function $g(m)$ is also extended to take a second argument d that gives the direction (FWD or BWD) in which the path should be traversed. FWD is the direction in which the path was created, while BWD refers to the reverse direction. Each router-specific function is extended in the same way.

2.4.2 Complex Processing

The current path model assumes that the path transformation is “work-preserving,” that is, for every input message, there is exactly one output message. This is limiting since it means that important operations such as packet reassembly and fragmentation cannot be accommodated. In the former case, most input messages do not result in an output message. Instead, the partial messages are buffered inside the router. In the latter case, every input message may result in many more than one output message. Similarly, a retransmission timeout may result in a

new message being generated spontaneously from within the path.

For this reason, we loosen the evaluation rule for paths. Suppose that creating a path results in the routers contributing the functions g_1, \dots, g_n . A message may now be injected at any one of these sub-functions and the invocation of g_i may result either in g_{i-1} or in g_{i+1} being invoked. That is, these sub-functions can be invoked in any order, subject to the rule that only neighboring functions are invoked, or that the message be enqueued at an output queue.

2.5 Remarks

In summary, a path is created incrementally by invoking a create operation on a router and specifying a set of invariants. The invariants describe the properties of the desired path, and are used to determine a next router that must be traversed by any message traveling on this path. The path reaches its maximum length when the invariants are no longer strong enough to make a unique routing decision. Each traversed router contributes a function g_i that is applied when processing a message. A path that traverses three routers is shown in Figure 5.

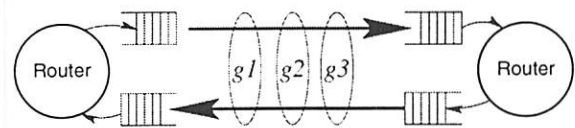


Figure 5: Example Path

Path execution is decoupled from the arrival and departure processes at the routers by four queues. For each direction, there is an input and an output queue. Typically, a path execution involves dequeuing a message from an input queue and evaluating the g_i functions in sequence until the other end of the path is reached. However, for generality, a message may get absorbed in the middle of a path, or turned around, or a new message may be created spontaneously inside a path.

Finally, keep in mind that policy issues—i.e., how to use paths for a given system—remain unspecified. There are two dimensions to this issue, which can be visualized as the “length” and the “width” of the path, respectively. The “length” of the path is simple to understand: it corresponds to the number of routers that the path traverses. A path’s width is more subtle: a highly specialized path is narrow, whereas a more general path is wide. For example, a path that can only be used to carry non-fragmented messages for a specific host-pair would be considered narrow, while a path from a network adapter to the IP protocol that can handle any IP datagram would be considered wide.

While it might seem that one wants paths to be as narrow (specific) as possible, this is not necessarily the case. Such a strategy can lead to an explosion of paths—e.g., one per packet or one per request/response transaction—which also implies having to create paths too frequently. Since there is a cost associated with path creation, one clearly wants the path to have enough breadth to carry multiple messages. The strategy we have adopted is to define a modest number of long-lived paths (e.g., one per window, one per open file, one per TCP connection) and then to define a small number of “short/fat” paths to catch the exceptional cases (e.g., all fragmented IP packets).

3 Implementing Paths in Scout

Scout is an experimental operating system designed for network appliances—e.g., set-top boxes, file- and web-servers, and cluster computers. Scout is designed around the path abstraction, supports both non-realtime and soft-realtime applications, and runs in a single address space. This section describes how the path abstraction is implemented in Scout.

Note that compatibility with standard application interfaces (e.g., POSIX) is not a major goal of Scout, except to understand how such interfaces either exploit or interfere with paths. On the other hand, interoperability with existing protocol specifications is an important requirement of Scout.

3.1 Routers and Services

Just as in the architecture, *routers* are the unit of program development in Scout. A router implements some functionality such as the IP protocol, the MPEG decompression algorithm, or a driver for a particular SCSI adapter. A router implements one or more *services* that can be used by other higher-level routers. As is typical in a layered system, most routers themselves use other lower-level routers to implement their services. Scout does not, however, enforce strict layering. Cyclic dependencies are admissible as long as there is a partial (non-cyclic) order in which the routers can be initialized.

Each service in a router has a name and a type. The names are unique, but otherwise arbitrary and chosen by the programmer. The relevance of service types is explained in more detail below. For the purpose of configuring a router graph, it is sufficient to know that two services can be connected by an edge only if they are mutually compatible. Figure 6 illustrates routers, services, and how they interact in a router graph. In this partial router graph, IP has three services: *up*, *down*, and *res*. The first two are of type *net* and the latter is of type *nsClient* (for “name-service client”). The *down* service is connected to

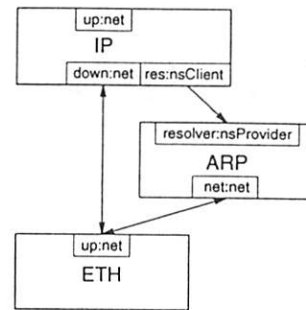


Figure 6: Routers and Services.

ETH’s *up* service. This connection is used by IP to send and receive IP datagrams. The *res* service is connected to ARP’s *resolver* service. IP uses this to translate IP host numbers into Ethernet addresses. ARP itself is connected to ETH as well so that it can broadcast and listen to relevant ARP packets.

A router is implemented simply as a collection of C source files. These files, along with the external interface, are described in a *spec* file. The syntax for spec files is shown below:

```

router name {
    files = {filename, ...};
    service = {name:type, ...};
}

```

A service name may be preceded by a less-than marker (<) to indicate that the routers connected to that service must be initialized before this router can be initialized. The Scout infrastructure ensures that router initialization occurs in an order that is consistent with the partial order defined by these markers. For the purpose of router initialization, cyclic dependencies are forbidden. The Scout development environment includes a configuration tool that translates a router graph into C source code that creates and initializes the runtime view of a router graph when the system boots. This configuration tool checks for and rejects any router graph with cyclic dependencies.

At runtime, a Scout router is represented by a variable of the following structure:

```

struct Router {
    String      name;
    long        (*init) (Router r);
    CreateStageFunc createStage;
    DemuxFunc    demux;
    RouterLinkList links[NSERVICES];
};

```

That is, a router consists of its name (member *name*), three function pointers (members *init*, *createStage*, and *demux*) and a list of router graph edges that connect to this router to other routers (member *links*).

Each router *r* provides just one globally visible operation:

```
Router rCreate (String n, int c[]);
```

This operation is used to create a specific router with name *n*. The integer array *c* specifies how many times each router service has been connected to other routers. Once all routers are created, Scout initializes them in the partial order described above. A router is initialized by a call to its *init* function.

3.2 Path Object

Section 2 argued that it is preferable to create paths incrementally, with the resulting paths initially consisting of a sequence of sub-functions *g_i*. Likewise, Scout paths consist of a sequence of *stages*. Each router that is crossed by a path creates one such stage. Since a path enters a router at one service and leaves it through another, a stage effectively connects a pair of services. That is, it represents a fixed routing decision.

A stage is a rich object that contains at least the following members:

```
struct Stage {
    Iface end[2];
    Path path;
    Router router;
    long (*establish)(Stage s, Attrs a);
    void (*destroy)(Stage s);
};
```

Member *end* is an array containing pointers to the interfaces of the stage. These interfaces are derived from the services that a stage connects in a manner that will be explained below. The *path* and *router* members point to the path that the stage is part of and the router that created the stage, respectively. The *establish* and *destroy* function pointers are used during path creation and destruction and are explained in more detail in Section 3.3.

The relationship between interfaces and router services is as follows. Each router service type consists of a pair of interface types: the first element in this pair specifies what interface the service *provides* whereas the second element specifies the interface that the service *requires* to function properly. For example, the *net* service type is symmetric in the sense that it both provides and requires a net interface. This can be expressed as the pair:

```
servicetype net = <NetIface, NetIface>;
```

Scout supports simple single inheritance for interface types. This means that instead of the exact interface type required by a service it is possible to provide a more specific interface. Hence, the precise rule used to decide whether a pair of services can be connected in a router

graph is that the interfaces provided must be identical to or more specific than the interfaces required.

All interfaces encountered when traversing a path in a particular direction are chained together. Since it is sometimes necessary to “turn around” the data flow inside a path, each interface also contains a back pointer to the next interface in the *opposite* direction. A third pointer provides access to the stage to which the interface belongs. Therefore, the most primitive interface is given by:

```
struct Iface {
    Iface next;
    Iface back;
    Stage stage;
};
```

This obviously is not a very interesting interface since it provides no way to deliver data. All real interfaces declare additional members that hold function pointers or other data. For example, the *net* interface is declared as follows:

```
struct NetIface {
    struct Iface i;
    long (*deliver)(Iface i, Msg m);
};
```

That is, the *net* interface provides a single function to deliver a message *m* to interface *i*. While Scout can technically support an arbitrary number of interface types, the intent is to keep this number as small as possible. For example, at present there is an interface type to asynchronously exchange messages (this is used both by filters and networking protocols), a window manager interface, a file system interface, and a few other, lesser interface types.

Given the definition of stages and interfaces, it is now easy to describe the actual path object:

```
struct Path {
    Stage end[2];
    long pid;
    void (*wakeup)(Path p, Thread t);
    PathQueue q[4];
    struct Attrs attrs;
};
```

The array *end* contains two pointers to the stages at the extreme ends of the path. A path can set the *wakeup* function pointer to request that a specific function gets executed when a thread *t* is awakened to execute in a path *p*. This is discussed more in Section 3.4. The four path queues are stored in *q*. These queues are generic in the sense that the queuing discipline is unspecified. The two properties that are defined for any such queue is the current length and the maximum length. Finally, *attrs* is a set of name/value pairs (attributes). Attributes allow to attach arbitrary state to a particular path. For example, this

enables stages to exchange and share information anonymously (without knowing exactly what stage is the source of the information and what stages are the consumers).

A path can therefore be visualized as shown in Figure 7. The path shown there consists of four stages. The stages were created by the TEST, UDP, IP, and ETH routers. Each interior stage contains two interfaces (semi-circles), whereas the stages at the extreme ends of the path contain only one interface each. These extreme stages are, strictly speaking, not part of the path but they are used to connect to the routers that manage the path queues.

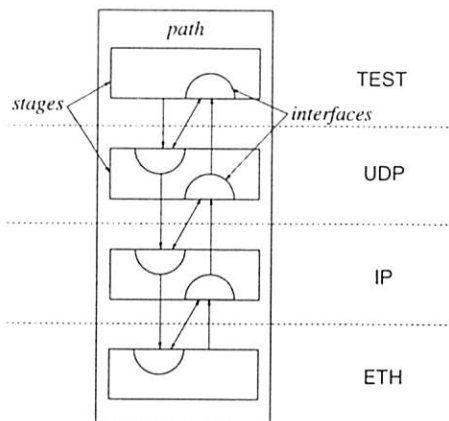


Figure 7: Path structure.

3.3 Path Creation

Paths are created and destroyed using the following functions:

```
Path pathCreate(Router r, Attrs a);
void pathDelete(Path p);
```

A path is created by invoking *pathCreate* on a router *r*. The kind of path to be created is described by the set of attributes *a*. These attributes are arbitrary name/value pairs that specify the invariants that hold true for the path being created. The *pathCreate* results in an invocation of the *createStage* function in router *r* (see Section 3.1). The *createStage* function has the following type:

```
Stage (*CreateStageFunc)(Router r, int s,
                          Attrs a,
                          RouterLink* n);
```

Here, *r* is the router on which *pathCreate* was invoked and *s* is the number of the service through which the path being created enters the router. Since *r* is the first router in the path, there is no such service, so the value is set to -1 (not a valid service number). Argument *a* is the set of attributes passed to the *pathCreate* operation. Once *r* creates a new stage and makes a routing decision, it sets *n* to

the router/service pair that the path must traverse next, if there is such a pair, otherwise it sets it to NULL.

Given the next router/service pair, the *createStage* operation is invoked on that next router. Now, argument *s* is set to the index of service through which the path enters the router and *a* is the (possibly modified) set of attributes. This process continues until a path reaches its full length, which happens either when it reaches a leaf router or when the attributes are so weak that no unique routing decision is possible. When either event occurs, a sequence of stages has been created. The stages and the interfaces contained therein are then linked together into a path structure. Once the path object is fully created, the *establish* functions in the stage objects are executed in the order in which the stages were created. This gives each stage a chance to perform initialization that depends on the existence of the entire path.

As described so far, path creation consists of three phases: (1) create sequence of stages, (2) combine stages into path object, and (3) establish (initialize) stages. During a fourth and final phase, path transformation rules are applied to the path. This provides the means through which Scout uses global knowledge to transform and optimize paths. Semantically, transformation rules have no effect, but they typically result in better performance and better resource allocation or usage. For example, if a path contains a sequence of interfaces for which there is optimized code is available, then the function pointers in the interfaces can be updated to point to this optimized code. More details on such code-related path-transformations can be found in a companion paper [23]. Section 4 discusses some transformations that improve resource management.

When a Scout system boots, there are typically a few routers that create a handful of paths, e.g., to receive key strokes or network packets. All other paths are either directly or indirectly created by these initial paths. In other words, path creation and destruction is under control of the routers that are present in a given system. The Scout infrastructure never creates or destroys paths implicitly.

3.4 Path Execution

Paths are executed by threads—the active entities in Scout. A router starts execution of a path by dequeuing data from the input queue and invoking an interface-type dependent data-delivery function.

Since threads are independent objects and since path queues can often be optimized away, it is possible for a thread to execute a path, enter a router, and then continue execution in another path without any context switches. This is important because degenerate paths can be short, so forcing context switches at every path/router crossing could result in an excessive number of context switches,

and therefore, less than optimal performance.

In Scout, threads are scheduled non-preemptively according to some scheduling policy and priority. Scout supports an arbitrary number of scheduling policies, and allocates a percentage of CPU time to each. The minimum share that each policy gets is determined by a system-tunable parameter. Two scheduling policies have been implemented to date: (1) fixed-priority round-robin, and (2) earliest-deadline first (EDF) [18]. The reason for implementing the EDF policy is that for many soft-realtime applications, it is most natural to express a path's "priority" in terms of a deadline. We present an example of this in the next section.

Scout uses a non-preemptive scheduler because it meets our needs and is easy to use. In the future, Scout will allow for uncooperative "threads," but since it is not a good idea to share *any* resource with uncooperative threads in an uncontrolled manner, those threads will not share memory either. That is, uncooperative threads will be isolated from each other in some manner (e.g., through separate address spaces, fault isolation, or a safe language). If uncooperative threads do not share memory, using a preemptive scheduler among them is trivial. Thus, scheduling is split into domains—within a domain, there is trust and hence a non-preemptive scheduler can be used. Across domains, there is no trust and a preemptive scheduler is necessary. This is not unlike what many traditional UNIX kernels do—the kernel "threads" are scheduled non-preemptively whereas the user-level processes are scheduled preemptively.

Once a thread executes on behalf of a path, it can trivially adjust its own priority as necessary. However, there also needs to be a mechanism that allows a newly awakened thread to inherit a path's scheduling requirements. For this purpose, a path can set the *wakeup* function pointer in its path object to a function that selects the appropriate scheduling policy and priority for a newly awakened thread.

3.5 Finding Paths

In many cases, knowing the path that should be used for a given set of data is trivial. For example, an application might create a path to a graphics window and then use that path to draw lines and paint text. In some cases, however, the path to be used is determined implicitly by the data itself. For example, when a packet arrives at a network adapter, it is not immediately known which path that packet belongs to. For this reason, each Scout router provides a *demux* operation that maps the data into a path that can be used to process that data. This problem is identical to what is referred to a "packet classification" in the networking literature. Since Scout uses packet classification in a context that is somewhat unusual, it is worth

enumerating the specific requirements that it places on this process:

- Efficient enough to handle peak-loads. Classification must take a short amount of time relative to the typical path execution time. Otherwise, the advantage of improved resource management due to paths would be lost.
- Provide relaxed (best-effort) classification accuracy. Unlike traditional classifiers, the Scout classifier just has to find a path that is "good enough" to process the given data. This is best illustrated with an example: suppose the data to be classified is an IP fragment. Traditional classifiers defer classification until the entire IP datagram has been reassembled. For the purposes of Scout, it is acceptable to hand off IP fragments to a path that knows how to reassemble the fragments. Once the full datagram is available, the IP protocol can rerun the classifier to find the next path.

Many packet classifiers have been proposed (e.g., [31, 20, 2, 8]), but none of them address all of the Scout requirements satisfactorily. For this reason, Scout adopted the simple solution of requiring each router to provide a function that performs a classification. Any given router typically implements only a small portion of the entire classification process. If a router cannot make a unique classification decision, it may ask the next router to refine that decision. This continues until either a unique path is found or until it is determined that no appropriate path exists. In the latter case the offending data is simply discarded.

3.6 Remarks

Figure 8 summarizes the Scout timeline. At the earliest time (top), individual routers and path transformations are implemented. Later on, a system is configured by specifying a router graph and selecting appropriate transformation rules. The kernel is then built and booted. During runtime, paths are created, executed, and eventually destroyed when no longer needed.

As implemented in Scout, paths are light-weight. For example, a path to transmit and receive UDP packets consists of six stages. Creating such a path on a 300MHz Alpha takes on the order of 200 μ s. This time does not include the application of any transformations. The path object itself is about 300 bytes long and each stage is on the order of 150 bytes in size (including all the interfaces). Also, packet classification is reasonably efficient. The first (unoptimized) implementation of the Scout classification scheme is already able to demultiplex a UDP packet in less than 5 μ s.

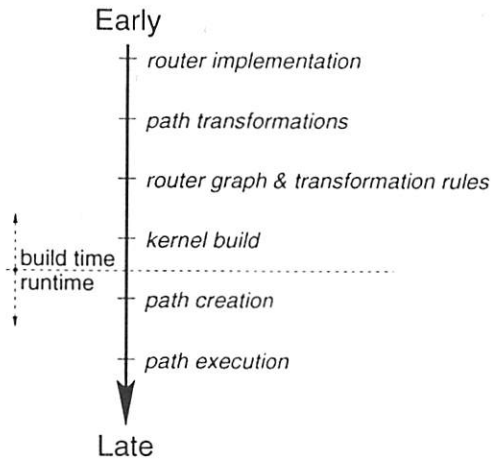


Figure 8: Scout Development Timeline.

There are many other aspects of Scout that space does not permit us to describe; most of them are orthogonal to paths. For example, we believe software-based fault isolation (SFI) [30] could be imposed on top of paths by defining each router to be in a separate fault domain. Similarly, hardware-enforced protection could be imposed between paths. Note that the horizontal partitioning (SFI) is possible because Scout routers have well-defined interfaces, while the vertical partitioning (hardware protection) is enabled by explicit paths.

Also, the Scout router graph is configured at build time, and as currently defined, it is not possible to extend the graph at runtime. However, it is possible to configure an interpreter into the router graph, thereby supporting extensibility. For example, we are currently implementing the Java API (and interpreter) in Scout [10]. This will make it possible to download Java applications into Scout at runtime.

4 Demonstration Application

This section demonstrates the use and benefits of paths with a simple, but realistic application implemented in Scout. The application consists of receiving, decoding, and displaying MPEG encoded video streams. MPEG encoding is able to reduce the size of a video by a factor of 10 to 100, but this compression ratio comes with a computationally expensive decompression algorithm. Workstations have only recently become fast enough to perform this task in realtime. Since MPEG decoding involves substantial computation, it is an application that demonstrates some of the advantages of paths related to resource management.

4.1 MPEG Router Graph

The Scout router graph for the demonstration application is shown in Figure 9. The topmost router, DISPLAY, manages the framebuffer. The bottom of the graph is formed by three routers implementing standard networking protocols: UDP, IP, and ETH. In the middle are the three interesting routers: MPEG, MFLOW, and SHELL.

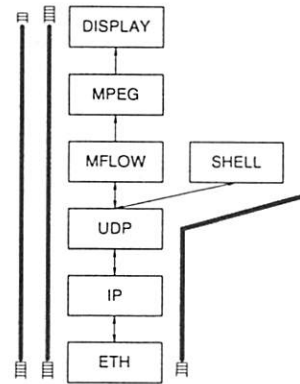


Figure 9: Router graph for MPEG example.

The MPEG router accepts messages from MFLOW, applies the MPEG decompression algorithm to them, and sends the decoded images to the DISPLAY router. There, the images are queued for display at the appropriate time. The MPEG router uses application-level framing (ALF) [4] to minimize internal buffering. That is, the MPEG source sends Ethernet MTU-sized packets that contain an integral number of work-units (MPEG macroblocks). This ensures that the MPEG decoder does not have to maintain complex state across packet boundaries and obviates the need for undesirable queueing between MPEG and MFLOW.

The MFLOW router implements a simple flow-control protocol. MFLOW advertises the maximum sequence number that it is willing to receive based on the sequence number of the last processed packet and the input queue size. MFLOW uses sequence numbers to ensure ordered, but not reliable, delivery of packets to MPEG.

The SHELL router is used to create paths dynamically. It is configured on top of UDP so it can receive command requests via the network. SHELL is not unlike a UNIX shell in that it waits for a command request which it then maps into a command "invocation." In the context of Scout, this involves mapping the command name into an appropriate path create operation. To create a path, SHELL requires two pieces of information: the router on which the path create operation is to be invoked on and a set of attributes (invariants). In the current implementation, an `mpeg_decode` command always results in a path create invocation on the DISPLAY router. In gen-

eral, SHELL might consult an environment variable to select the graphics display to be used. SHELL creates MPEG paths with the following two attributes:

PA.NET.PARTICIPANTS=(*ip-addr, udp-port*):

This attribute specifies the network address of the process that sent the `mpeg_decode` command request. SHELL assumes that the network address of the video source is the same as the address that originated the command request.

PA.PATHNAME="MPEG": The value of this attribute is a string that, in its simplest form, is interpreted as a sequence of router-names. It is used either to force a specific routing decision or to supply routing information when there is no other routing information available. In the case of an MPEG path, SHELL sets this attribute to the string "MPEG" to force DISPLAY to forward path creation to the MPEG router.

Another attribute that is used during MPEG path creation is **PA.PROTID**. Unlike the other attributes, this one is not specified by the SHELL router. Instead, it is reset by each router that implements a networking protocol. The value of this attribute is the protocol id of the next-higher level networking protocol. This id is normally needed during packet classification. For example, IP packets with a protocol type of 6 are TCP packets and TCP packets with a port number of 21 are normally FTP packets. So when FTP forwards a path create operation to TCP, it sets **PA.PROTID** to 21. If TCP decides to forward path creation to IP, it resets the value of **PA.PROTID** to 6 to let IP know that it is dealing with a TCP path.

Figure 9 shows two video paths (from ETH to DISPLAY) and a shell path for receiving commands (from ETH to SHELL). Note that the video paths take their input from, and deposit their output into, a queue. These queues are serviced by interrupt handlers. In ETH, the queue is filled in response to a receive interrupt, and in DISPLAY, the queue is drained in response to the vertical synchronization impulse of the video display. Output to the display is synchronized to this impulse because there is no point in updating the display at a higher frequency.

There are three points worth emphasizing about this example. First, there are no queues other than the ones in ETH and DISPLAY. As mentioned above, this is due to MPEG's use of ALF. Second, ALF—along with explicit paths—enable integrated layer processing. Since MPEG reads the network packet data in units of 32 bits, it would be straight-forward to integrate the (optional) UDP checksum with the reading of the MPEG data. This would require a path-transformation rule that matches for MPEG being run directly on top of UDP. If this pattern matches, the path can be transformed by replacing

the UDP and MPEG receive processing functions with functions that implement the UDP checksum computation as part of MPEG's reading of the packet data. Third, without queuing in the middle of the path, scheduling is simplified—if the output queue is full already, there is little point in scheduling a thread to process a packet in the input queue. This implication would not hold in the presence of additional queues.

Table 1 gives measurements that indicate the performance a Scout MPEG kernel can achieve. The table lists the maximum decoding rate in frames per second for a selection of four video clips. To put these numbers in perspective, the table also gives the corresponding numbers for Linux. The numbers are comparable in the sense that both systems run on the same machine (a 300MHz 21064 Alpha), use essentially the same MPEG code, and receive the compressed video over the network. The dominant costs in this example are the decompression of the MPEG stream and the dithering and displaying of the video frames. That is, practically all time is spent in the MPEG and DISPLAY routers.

Video	# of frames	max. rate [fps]	
		Scout	Linux
Flower	150	44.7	37.1
Neptune	1345	49.9	39.2
RedsNightmare	1210	67.1	55.5
Canyon	1758	245.9	183.3

Table 1: Coarse-Grain Comparison of Scout and Linux

While the playing field was as level as we could make it, it must be understood that this is an apples and oranges comparison—the two systems have a very different scope, level of functionality, and maturity. Still, the comparison is useful to establish that a path-based system such as Scout can easily achieve performance that is consistent with the machine on which it runs.

4.2 Queues

As Figure 9 shows, two queues exist at the ends of the MPEG path. These queues are in the ETH router (the input queue) and in DISPLAY (the output queue).

The input queue is required for two reasons: (1) for high-latency networks it may be necessary to have multiple network packets in transit, and (2) because of network jitter, these multiple packets may all arrive clustered together. Since the peak arrival rate at the Ethernet is much higher than the MPEG processing rate, the queue is needed to absorb such peaks.

Whereas the input queue absorbs bursts that are limited in size, the job of the output queue is to absorb jitter

at a more global level—decompression itself introduces significant jitter. Depending on the spatial and temporal complexity of a video scene, the encoded size of any particular video frame may be orders of magnitudes different from the size of the average frame in that stream. The network may also suffer from significant jitter, e.g., due to temporary congestion of a network link. Finally, the sender of the MPEG stream itself is likely to add jitter since the video may, for example, be read from a disk drive. Just how big should these queues be? Obviously, they should be “just big enough,” but is it possible to put some quantitative limits on their sizes?

First, consider the input queue. If processing a single packet requires more time than it takes to request a new packet from the source, then an input queue that can hold two packets is sufficient: one slot is occupied while the last received packet is being processed, and the second (free) slot is advertised to the source. If the round-trip time (RTT) is greater than the time to process a packet, then the input queue needs to be two times the $RTT \times$ bandwidth product of the network. MFLOW can measure the round-trip latency by putting a timestamp in its header. The important point from the perspective of this paper, however, is that accurate measurement of the peak processing rate is enabled by paths—it is a simple matter of specifying the appropriate transformation rule to ensure that the average time spent processing each packet is measured. For MPEG, this means that the initial function in the ETH-stage of the router is modified to measure processing time and to update the path attribute that keeps track of the average processing time.

In the case of the output queue, the factors influencing queue size are more varied and complex. A complete analysis is beyond the scope of this paper. In general, bounding the size of this queue requires cooperation with admission control and would typically employ a network reservation system, such as RSVP [3]. The current implementation leaves this parameter under user control to facilitate experimentation.

4.3 Scheduling

Since each video path has its own input queue and since the packet classifier is run at interrupt time, newly arriving packets are immediately placed in the correct queue. This means that once a packet is under control of the software, there is no danger of priority inversion due to low-priority packets being processed ahead of high-priority packets. This is one of the most significant advantages of paths. For example, the early separation makes it possible to run a video stream while flooding the network adapter with small Ethernet packets.

This is demonstrated in Table 2, which shows how the maximum decoding frame rate for the Neptune video

drops when load is added to the Scout and Linux systems, respectively. The additional load consists of a flood of ICMP ECHO requests (generated with `ping -f`). In the Scout case, the video path is run at the default round robin priority, whereas the path handling ICMP requests is run at the next lower priority. In contrast, Linux handles ICMP and video packets identically inside the kernel. As the table shows, adding the ICMP load has little effect on the frame rate for Scout, while the maximum framerate for Linux drops by more than 42%. Clearly, the early separation afforded by paths can have significant benefits. This is not to say that paths are the only way to solve this particular problem (e.g., [22]), but it does support our claim that paths can be an effective solution to such problems.

	Framerate [fps]		
	unloaded	loaded	Δ
Scout	49.9	49.8	-0.2%
Linux	39.2	22.7	-42.1%

Table 2: Frame Rate Under Load

While the advantages of paths due to early separation are important, paths play an even more intimate role in scheduling. As explained in Section 3, a path can register a wakeup callback that can be used to adjust a thread’s scheduling policy and priority according to its own needs. The MPEG path uses this facility to ensure that any thread that is ready for execution in the path will be scheduled with the proper realtime constraints. In combination, separate input queues and proper scheduling guarantee that the MPEG Scout kernel has no difficulty in delivering and processing realtime MPEG packets even under severe background loads. For example, an arbitrary number of low-priority MPEG streams (or some other non-realtime background work) can be displayed without adversely affecting realtime streams running in the foreground.

The default Scout scheduler is a fixed-priority, round-robin scheduler. Since video is periodic, it seems reasonable to use rate-monotonic (RM) scheduling for MPEG paths. With RM scheduling, a (periodic) realtime thread receives a priority level that is proportional to the rate at which it executes. That is, the frame-rate at which a video is displayed would control the priority of the corresponding path. However, there are several reasons that make earliest-deadline-first (EDF) scheduling more attractive than RM scheduling. These include:

- The frame-rate must be under user control to support features such as slow-motion play or fast forward. This implies that a large number of priority-levels would be necessary. Otherwise, two MPEG

paths that have similar, but not identical, frame-rates could not be distinguished scheduling-wise. If the number of priority levels is large, EDF scheduling is just as efficient as RM scheduling.

- MPEG decoding is periodic, but not perfectly so. Consider playing a movie at 31 Hz on a machine with a display update frequency of 30 Hz. Given that only 30 images can be displayed every second, it will be necessary to drop one image during each one second interval. When the drop occurs, there is no need to schedule that path, so a fixed priority would be sub-optimal.
- While not a quantitative argument, probably the strongest case for EDF scheduling is that it is the *natural* choice for a soft realtime thread that moves data from an input queue to an output queue. For example, if the output queue drains at 30 frames/second and the queue is half full, it is trivial to compute the deadline by which the next frame has to be produced.

For these reasons, Scout uses EDF scheduling for real-time MPEG paths. For example, this allows Scout to display 8 Canyon movies at a rate of 10 frames per second, together with a Neptune movie playing at 30 frames per second, all without missing a single deadline. In contrast, the same load with single-priority round-robin scheduling leads to a large number of missed deadlines if the output queues for the Canyon movies are large.³ For example, with a queue size of 128 frames, on the order of 850 out of 1345 deadlines are missed by the path displaying the Neptune movie. The reason for the poor performance of round-robin scheduling is that it keeps scheduling the 8 Canyon movies as long as their output queues are not full, even at times when the Neptune movie needs the CPU much more urgently.

One question that remains is how the deadline is computed. Here again, paths play a central role. If path execution is the bottleneck, then the output queue should be kept as full as possible. In this case, it is best to set the deadline to the display time of the next frame to be put in the output queue. In contrast, if network latency is the bottleneck, then the deadline should be based on the state of the input queue. Since at any given time some number of packets (n) should be in the transit to keep the network pipe full, MFLOW must be able to advertise an open window of size n . This means that the deadline is the time at which the input queue would have less than n free slots. This time can be estimated based on the current length of the queue and the average packet arrival rate.

³Because it is difficult to compute globally meaningful priorities for RM scheduling in a dynamic system—i.e., one where different rate videos come and go—single-priority round-robin is the next best alternative. Therefore, we compare EDF to this case, rather than to RM.

Since the path object provides direct access to both queues, the effective deadline can simply be computed as the minimum of the deadlines associated with each queue. Alternatively, the path can use the path execution time and network round-trip time to decide which queue is the bottleneck queue, and then schedule according to the bottleneck queue only. The latter approach is slightly more efficient, but requires a clear separation between path execution time and network round-trip time. The implemented MPEG decoder is currently optimized for the case where the output queue is the bottleneck, so scheduling is always driven off of that queue.

4.4 Admission Control

Finally, paths enable admission control. As all memory allocation requests are performed on behalf of a given path, it is a simple matter of accounting to decide whether a newly created path is admissible or not. Before starting path creation, the admission policy decides how much memory can be granted to a new path. As long as each router in the path lives within that constraint, the path creation process is allowed to continue. (Note that admission control has not yet been implemented in Scout.)

Paths are also useful in deciding admissibility with respect to CPU load. Again, this is due to the fact that it is easy to compute the execution time spent per path—our experiments show that there is a good correlation between the average size of a frame (in bits) and the average amount of CPU time it takes to decode a frame. Naturally, the model that translates average frame size into CPU processing time is parameterized by the speed of the CPU, the memory system, and the graphics card. Rather than determining these parameters manually, it is much easier to measure path execution time in the running system and use those measurements to derive the required parameters. That is, the path execution timings are used to derive the model parameters, which in turn, are used for admission control.

Finally, if admission control determines that a video cannot be displayed at the full rate, a user may choose to view the video with reduced quality. For example, the user may request that only every third image be displayed. Thanks to ALF and paths, it is possible to drop packets of skipped frames as soon as they arrive at the network adapter. This avoids wasting CPU cycles at a time when they are at a premium.

5 Related Work

At a superficial level, Scout paths are similar to UNIX pipes [28] and Pilot streams [25]. While all three abstractions have in common a linear sequence of “components”

(processes in UNIX, Mesa modules in Pilot, stages in Scout), neither pipes nor streams provide any global context to the individual modules. Neither do they attempt to optimize the code along a “path.” It is also the case that UNIX pipes are more coarse-grain and uni-directional.

As mentioned in Section 1, there is a wealth of mechanisms that offer point-solutions to the more general problem of exploiting paths, both as a structuring framework and as an optimization technique. This related work falls broadly into two categories, depending on their primary objective:

- optimizing code along the “fast path,” or
- improving resource management.

Examples of fast path optimizations include Synthesis [19], Synthetix [24], PathIDs [17], Protocol Accelerators [29], and integrated layer processing [4, 1]. Examples in the second category include processor capacity reserves [21], distributed/migrating threads [5, 9], and Rialto activities [16]. Because space does not permit us to contrast all of this work in detail, we simply point out that the path abstraction as presented in this paper is an attempt at unifying these various ideas. In particular, the proposed abstraction allows us to reason about both the fast path and resource management issues. Our claim is that the unifying principle behind this abstraction is the global knowledge that paths afford. The rest of this section discusses the related work we consider most relevant in more detail.

The system that is probably closest to Scout, at least in terminology, is Da CaPo (dynamic configuration of protocols) [11]. It defines an infrastructure for building multimedia protocols. While Da CaPo has a notion of paths and stages, there are important differences at all levels. At lowest level, Da CaPo paths are uni-directional and stages have very restricted functionality (they are essentially non-blocking event-handlers). As a result, Da CaPo is just powerful enough to accommodate common networking tasks. Also, interoperability with existing protocols is not a goal of Da CaPo. Another important difference is that path creation is left completely to an external “configuration manager.” As pointed out in Section 2, this means that in any reasonably complex system, the configuration manager will be burdened by detailed knowledge of the internal workings of particular protocols. In contrast, our path abstraction makes it easy to exploit both local and global knowledge during path creation. At a higher level, Da CaPo focuses completely on *automatically selecting* appropriate protocol functionality; performance and resource allocation appear to be secondary issues.

Kay [17] introduces the notion of a PathID, which is designed specifically to reduce the latency of receive-side network processing. Fundamentally, a PathID is similar

to a fine-grained virtual circuit identifier (VCI) in ATM networks [7]. Since PathIDs are stored in a known location in the header of network messages, packet classification becomes trivial (in the worst case a table-lookup). In [17], packets with a PathID are processed by highly optimized, handwritten code. Since this code is manually tuned, maintainability and ease of use are problematic. In fact, the paper suggests that PathIDs should be used for the rare cases where having to maintain two parallel branches of source code is a justifiable cost. Finally, PathIDs do not attempt to elevate paths to a fundamental OS structure, and the problem of creating paths without human direction is not addressed.

6 Concluding Remarks

This paper makes two contributions. First, it describes how paths can be made an explicit OS abstraction, and shows how this abstraction has been implemented in the Scout operating system. Second, it makes a case for why paths should be made explicit. This case includes both the intuitive arguments made in Section 1, and a demonstration of how paths proved beneficial in one particular application—receiving, decoding, and displaying MPEG-compressed video. On this latter point, we showed how paths are used to:

- segregate work early to avoid priority inversion;
- schedule the *entire* processing along a path according to the bottleneck queue, and to automatically determine the bottleneck queue in the system;
- provide accountability to decide the admissibility of a memory allocation request; and
- discard unnecessary work early to minimize the waste of resources.

What remains to be done is to demonstrate Scout—and the utility of paths—on a wider set of domains. For example, work on a Scout-based Java-box, active network router, and scalable storage server are under way.

Acknowledgments

We would like to thank the other members of the Scout group, particularly, John Hartman, Brady Montz, Patrick Bridges, David Larson, and Rob Piltz. Thanks also to the reviewers, especially our shepherd, Rich Draves. This work supported in part by DARPA Contract DABT63-95-C-0075 and NSF grant NCR-9204393.

References

- [1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.
- [2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 115–123, 1994.
- [3] R. Braden, D. Clark, and S. Shenker. RFC-1633: Integrated services in the Internet architecture: an overview. Available at <ftp://ftp.internic.net/rfc>, July 1994.
- [4] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of SIGCOMM '90 Symp.*, pages 200–208, Sept. 1990.
- [5] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, Apr. 1993.
- [6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 189–202, Dec. 1993. ACM.
- [7] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proc. of SIGCOMM '94 Symp.*, pages 2–13, Aug. 1994.
- [8] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. of SIGCOMM '96 Symp.*, pages 53–59, Aug. 1996.
- [9] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *1994 Winter USENIX Conf.*, pages 97–114, Jan. 1994.
- [10] J. Gosling, F. Yellin, and The Java Team. *The Java Application Programming Interface*. Addison-Wesley, Reading, MA, 1996.
- [11] A. Gotti. The Da CaPo communication system. Technical report, ETHZ, Switzerland, June 1994.
- [12] A. Habermann, L. Flon, and L. Cooprier. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.
- [13] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conf.*, pages 147–159, June 1993.
- [14] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.
- [15] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.
- [16] M. Jones, P. Leach, R. Draves, and J. Barrera. Support for user-centric modular real-time resource management in the Rialto operating system. In *Proc. of the 5th Intl. Workshop on Network and OS Support for Digital Audio and Video*, pages 55–66, Apr. 1995. ACM.
- [17] J. S. Kay. *Path IDs: A Mechanism for Reducing Network Software Latency*. PhD thesis, University of California, San Diego, 1995.
- [18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):46–61, Jan. 1973.
- [19] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, Sept. 1992.
- [20] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *1993 Winter USENIX Conf.*, pages 259–269, Jan. 1993.
- [21] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proc. of the 4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 129–134, Oct. 1993.
- [22] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *1996 Winter USENIX Conf.*, pages 99–112, Jan. 1996.
- [23] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of techniques to improve protocol latency. In *Proc. of SIGCOMM '96 Symp.*, pages 73–84, Sept. 1996.
- [24] C. Pu et al. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. of the 15th ACM Symp. on Operating System Principles*, pages 314–324, Dec. 1995.
- [25] D. D. Redell et al. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.
- [26] R. V. Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, pages 80–89, Aug. 1995.
- [27] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct. 1984.
- [28] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [29] R. van Renesse. Masking the overhead of protocol layering. In *Proc. of SIGCOMM '96 Symp.*, volume 26, pages 96–104, Stanford, CA, Aug. 1996. ACM.
- [30] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 203–216, Dec. 1993.
- [31] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *1994 Winter USENIX Conf.*, pages 153–165, 1994.
- [32] H. Zimmermann. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, Apr. 1980.

Studies of Windows NT Performance using Dynamic Execution Traces

Sharon E. Perl and Richard L. Sites
Digital Systems Research Center
Palo Alto, CA
{perl,sites}@pa.dec.com

Abstract

We studied two aspects of the performance of Windows NTtm: processor bandwidth requirements for memory accesses in a uniprocessor system running commercial and benchmark applications, and locking behavior of a commercial database on a small-scale multiprocessor. Our studies are based on full dynamic execution traces of the systems, which include *all* instructions executed by the operating system and applications over periods of a few seconds (enough time to allow for significant computation). The traces were obtained on Alpha PCs, using a new software tool called PatchWrx that takes advantage of the Alpha architecture's PAL-code layer to implement efficient, comprehensive system tracing. Because the Alpha version of Windows NT uses substantially the same code base as other versions, and therefore executes nearly the same sequence of calls, basic blocks, and data structure accesses, we believe our conclusions are relevant for non-Alpha systems as well. This paper describes our performance studies and interesting aspects of PatchWrx.

We conclude from our studies that processor bandwidth can be a first-order bottleneck to achieving good performance. This is particularly apparent when studying commercial benchmarks. Operating system code and data structures contribute disproportionately to the memory access load. We also found that operating system software lock contention was a factor preventing the database benchmark from scaling up on the small multiprocessor, and that the cache coherence protocol employed by the machine introduced more cache interference than necessary.

1 Introduction

This work was triggered by two performance puzzles (circa 1995) related to Microsoft's SQL Server running on Alpha [SW95] PCs under the Windows NT operating system: how could we speed up the uniprocessor ver-

sion, and how could we get closer to linear scaling for the multiprocessor version?

To answer these questions we found that we needed to look at the detailed behavior of the system under load. We created a tool for obtaining complete traces of the instruction and data streams executed by the processor in all operating system and application code. We proceeded to examine these traces and to use them to run simulations that revealed interesting properties of the then current system. The results of the simulations also had implications for future processor design.

The first puzzle we studied is how the processor bandwidth requirements of applications—the bandwidth required to service on-chip instruction and data cache misses from off-chip caches or memory—limits the achievable execution speed of the applications. This study was motivated by a discussion with some colleagues about using a prefetching strategy to improve performance. Their studies showed that prefetching would not help in the particular situation because there wasn't enough processor-chip pin bandwidth to support the workload. When pin bandwidth is a bottleneck, some common techniques for trying to improve performance do not help. These include multiple instruction issue, code scheduling, prefetching, and improved external cache latency. Pin bandwidth puts a ceiling on how fast an application can run. We discovered that pin bandwidth is indeed a bottleneck for interesting commercial applications, as well as for data accesses in one of the SPEC benchmarks that we studied.

The second puzzle we studied is how lock contention for a multiprocessor application limits the scalability of the application. Detailed execution traces reveal patterns of locking that may not have been expected by the operating system designers, application designers, or hardware designers responsible for the cache coherence protocol. Lock contention and the related cache coherence overhead prevent the application from scaling up beyond a small number of processors.

We choose these two aspects to study because they

seem to be important factors in overall workload performance.

The contributions of this work are threefold: we provide evidence of processor pin bandwidth limitations and locking problems for commercial applications; we introduce a new tool for obtaining full traces of a system that allows us to study such problems; and we are making available some of the traces upon which our studies are based for other researchers to use.¹ Also, the understanding we gained from this work led to significant improvements to the hardware and software involved, so the results presented here do not fully apply to currently shipping hardware and software.

In the next section we describe some of the more interesting aspects of the tracing tool, called PatchWrx. Section 3 describes our studies of pin bandwidth requirements for four different applications on two different Alpha processors. Section 4 describes studies of locking behavior of one of these applications—the Microsoft SQL Server database—on a small multiprocessor. Section 5 concludes.

2 PatchWrx

To understand the traces that are the input to our performance studies it is helpful to understand the properties of PatchWrx, the tool used to produce them. In this section we give an overview of PatchWrx, and then describe some of the highlights of its design and implementation.

2.1 Overview

PatchWrx is a software-only technique for capturing full time-stamped traces of the dynamic instruction stream (i-stream) and data stream (d-stream) of operating system and user code.

The goal of the PatchWrx effort is to capture traces of every single instruction executed throughout many seconds of a real operating system, running some complex workload on a non-microprogrammed multiprocessor. We wanted to build a software-only solution, rather than requiring one-of-a-kind hardware that cannot easily be applied at a customer computing environment. We wanted to gather traces with less than a factor of ten performance degradation, so that nothing in the operating system broke due to timeouts or excessive latencies. Finally, we wanted to work with arbitrary binaries for which we did not have access to the source code.

The technique we have adopted rewrites binary executable images, inserting patches that record in a log the

¹Our uniprocessor traces are publicly available on CD-ROM to full-time faculty members. Contact one of the authors to obtain a copy.

target of every change of control flow (branch, call, return, system call, system return, interrupt, and return-from-interrupt), and some base register values for d-stream memory accesses. A reconstruction program working from the log and binary program images reproduces the trace of the full i-stream and d-stream that was executed. Our logs are typically about 5–10 times smaller than the resulting traces.

With the entire operating system patched for just branches (not load/store) and logging on, everything runs at about 1/4 of normal speed until the log buffer fills up. Then logging is turned off and the run speed is about 1/2 of normal speed. This is sufficiently fast that our personal machines have run patched all the time for over a year. With loads and stores patched in the operating system and applications, the worst slowdown we've seen is about 1/8 of normal speed. Patched images are 30–50% larger than the originals.

PatchWrx is an offshoot of the ATUM work in tracing using microcode [ASH86], and work with binary translation [SCK⁺93]. Our approach is similar to other inline tracing efforts, but differs significantly in at least one dimension. Most published studies are of user code only [EKKL90, LB94], or are done on a single processor [BKW90, CB93], or require rebuilding source code [SJF92], or trace only cache misses, not all instructions [CHRG95, TGH92]. None use Windows NT. The excellent Shade paper [CK94] summarizes about thirty previous tools. Using that paper's classification, PatchWrx, like ATUM, traces executables, user and system code, multiple domains, multiple processors, signals, dynamic linking, and bugs, with performance similar to Shade.

We chose to produce traces rather than do on-the-fly data analysis [SE94] because of the difficulty of recreating complex execution environments months after the original investigation. With a detailed trace, questions asked a year or more later can still be investigated.

Somewhat like an electron microscope for computing, the PatchWrx approach is for studying a small amount of execution in excruciating detail, rather than summarizing long-running executions.

All of our experiments have been performed under Windows NT, version 3.5. The uniprocessor experiments were run on an Alpha AXP 150 with 128 MB of main memory. The multiprocessor experiments were run on a four-processor AlphaServer 2100, with 190 MHz processors and 256 MB of memory.

2.2 Trace Contents

The final output of PatchWrx is a trace containing the sequence of instructions executed by the operating system and all applications from the time logging was enabled

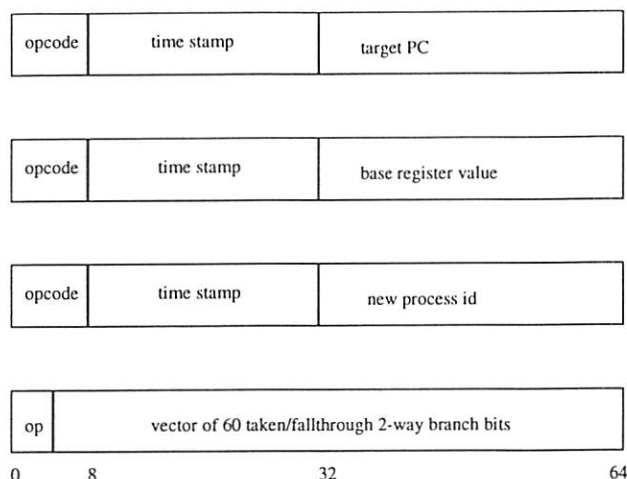


Figure 1: Formats of PatchWrx log entries. Each entry is 64 bits.

up until the log buffer (from which the trace is reconstructed) filled up.

Each instruction in a trace is tagged with its program counter value, and if it is a load or store instruction, the memory virtual address that is its source or target. In a multiprocessor trace, the instruction is also tagged with its processor number.

Some versions of our traces also contain timestamps on all branching instructions except for two-way branches. These are used to help line up the i-streams from different processors in the multiprocessor traces, and to compute lock holding times in our second study, described in Section 4.

2.3 Log Entries

While a patched system is running, a log is collected recording the information necessary to eventually reproduce a full system trace. A log is a sequence of entries describing branching events or data references. Figure 1 summarizes the different kinds of log entries.

As in most computer designs, the Alpha architecture has two basic forms of branching instructions: a jump to an address in a register, and a PC-relative two-way conditional branch. For each jump instruction, we record one log entry. For two-way branches, we accumulate a bit-vector recording the outcome of up to 60 two-way branches in a single log entry, one bit per branch. This gives us a compact encoding of the exact flow within a subroutine, taking only about 10% of the log entries.

When an interrupt or page fault occurs during logging, the address of the first instruction of the interrupt or fault handler is put in the log, along with the address of the first instruction *not* executed because of the inter-

rupt (the instruction that would normally be executed immediately after the interrupt handler returns). This information is used during trace reconstruction to determine exactly where the interrupt occurred in the i-stream.

Additional log entries record information about memory load and store instructions, and process context switches.

In the Alpha implementation of PatchWrx, the log is recorded in a 45 MB portion of physical main memory that is reserved at boot time, and is therefore invisible to the operating system. The log buffer holds about 5.9 million eight-byte log entries, which is enough for 5–20 seconds of real time. There is so much information in a single reconstructed trace that we have not been motivated to try stitching multiple traces together [CHRG95, AH90]; a single reconstructed trace contains about 650 MB of dynamic i-stream with instruction and data addresses.

Recording the log in main memory is much faster than recording on disk or tape. Recording in physical memory instead of virtual memory allows us to trace the lowest levels of the operating system, including the page-fault handler, without generating recursive page faults. It also allows us to trace across multiple threads running in multiple address spaces, without needing to write a log entry to one address space while executing in a different address space.

2.4 PAL Subroutines for Logging

To implement the logging code, we use the Alpha architecture's PALcall instruction, which traps to one of a set of Privileged Architecture Library (PAL) specialized subroutines without disturbing any programmer-visible state, such as registers. These subroutines have access to physical main memory and to internal hardware registers and they run with interrupts turned off. We extended the PAL-code for Alpha NT with eight additional subroutines, and we modified some of the existing subroutines, as summarized in Table 1. Other architectures may have supervisor call or trap instructions that, in conjunction with modified operating system kernel interrupt routines, could be used to get a similar effect.

2.5 Collecting Data Addresses

It is possible to capture data addresses by patching all load and store instructions, but this fills up the log buffer quite quickly and so we would like to avoid it. We observe that many pieces of code use multiple references off the same base register. Since the i-stream reconstruction recovers the actual instructions executed, we will have these address specifications in the dynamic i-stream. Thus, for a sequence of references over which

PAL routine	Action/Recorded info.
PalReset	(Set aside log memory)
InterruptStackDispatch	next addr., interrupt target
SoftwareInterrupt	next addr., interrupt target
DispatchMmFault	next addr., page fault target
UNALIGNED	next addr., align. fault target
RFE	return from exception target
CALLSYS	sys. call target
RETSYS	return from sys. call target
SWPCTX	new process ID
pwrden	read log entry from buffer
pwctrl	init. log, turn logging on/off
pwbsr	branch entry
pwjsr	jump/call/return entry
pwldst	load/store base register entry
pwbrt	cond. branch taken
pwbrf	cond. branch fall-through
pwpeek	(for debugging only)

Table 1: Logging-related PAL subroutines. First set are modifications to existing PAL subroutines. Second set (starting with pwrden) are new PAL subroutines for PatchWrx.

changes to the base register value can be computed from the i-stream, we need only record the base register's value once. The effect is that only one out of every 5–10 load or store instructions is actually patched, and that for loops with constant strides through memory, only the initial base-register value outside the loop is traced.

Our patching and reconstruction algorithms are somewhat simplistic, causing the reconstruction to reach some loads and stores without knowing the base register value (as with certain interrupts). When this happens during reconstruction, we make up a random synthetic value for the base register, then track any incremental changes from there. Even in code with no load/store patches at all, this is surprisingly useful. The effect seen in the d-stream is that the bases of arrays and structures and stacks are random, but the relative access patterns within each aggregate are accurately reflected.

2.6 Handling Multiprocessors

When PatchWrx is running on multiple processors, the log entries for all processors are merged into a single log buffer. This allows us to see the dynamics of the interactions between processors. Writing a single merged log requires doing a multiprocessor-atomic update of the next-log-entry pointer, and requires encoding the processor number in each log entry.

Rather than interleaving single entries, we allocate chunks of 128 entries to each processor. This cuts down the frequency of atomic updates by two orders of mag-

nitude and lets us encode the processor number once per chunk of entries, rather than in every entry. With log entries generated on each processor at the rate of about one per microsecond, a group of four chunks represents about 100 microseconds of real time on a four-processor system.

2.7 Trace Reconstruction from Logs

Reconstruction of a full trace given a log and a set of binaries is mostly straightforward. As described above, some special care and techniques are applied for obtaining data addresses from a limited set of base register values recorded in a log. Two additional tricky issues involve handling interrupts and merging multiprocessor traces.

The first issue concerns where to insert interrupts in a reconstructed i-stream. During reconstruction, one potential place to divert the i-stream to the interrupt handler is after all jump entries that precede the interrupt have been consumed, when the instruction that matches the not-executed address in the next log entry is encountered. If the not-executed instruction is inside a loop, the interrupt must be delivered during the right iteration. Loop iterations are controlled by jumps or conditional branches. So it is necessary to consume not only all the preceding jump entries in the trace, but also exactly the right number of conditional branch bits before delivering the interrupt. For this reason, we flush the partially accumulated taken/fallthrough vector into the log buffer just before recording the trace entries for an interrupt. This allows a perfect reconstruction of where to deliver an interrupt.

The second issue concerns merging traces from multiple processors. This requires special care because the time stamps within the entries come from four different cycle clocks (oscillators) on a four-processor system. These clocks are not synchronized with each other, and we observe drift of up to 100 parts per million (100 microseconds per second) within logs. All we really know about the clocks is that the first time stamp in chunk N was created after the first time stamp in chunk $N - 1$ and before the first one in chunk $N + 1$.

By carefully applying running inequalities between hundreds of chunks, we can map the drifting time bases into a single absolute time from the beginning of the log, within a window about two microseconds wide. We use this derived absolute time base to interleave the instructions in the reconstructed merged i-stream. The effect is that an instruction on one processor that stores to a shared variable is close (within dozens of instructions at a 4x tracing slowdown) in the final i-stream to an instruction on another processor that reads the shared variable and in fact sees the new value. Although we cannot guarantee

that the store precedes the load in the final i-stream, we find this close enough for understanding multi-processor dynamics, including cache interactions.

2.8 I-stream Distortion

Because the patches introduce extra instructions and extra memory references on an instrumented system, there is necessarily some distortion in the timing of the reconstructed i-stream. There is little distortion of the i-stream itself, since the reconstruction excludes the patches. The paths through subroutines and the sequence of subroutine calls are the same in the reconstructed i-stream as they would have been in an uninstrumented system.

The reconstructed and uninstrumented i-streams differ in four subtle ways, however.

First, the patched images are bigger than the originals, so shared-library images loaded into consecutive memory locations end up at somewhat different addresses. This can have a slight effect on instruction cache (i-cache) simulations for large caches based on the traced instruction addresses. Patching expands images by multiples of 64 kilobytes, so the larger size images have no effect on small cache simulations; the low 16 bits of patched and original i-stream addresses are identical. There can be a similar slight effect on data cache (d-cache) and translation buffer simulations, as patches can also cause data to be moved.

Second, the patched images can take i-stream page faults on the pages containing the patches. While the patches themselves never show up in the reconstructed i-stream, the extra page fault traps and processing do. This appears to be only a slight distortion.

Third, timer interrupts happen four times more frequently in the traced code than in the uninstrumented code, due to the 4x tracing slowdown. In Windows NT 3.5, this increases the total number of instructions executed by about 1 percent. We have chosen to ignore this in our studies, but one could mechanically remove three of every four timer interrupts to further reduce the distortion.

Finally, external interrupts (disk, network, etc.) happen approximately four times sooner in the traced code than in the uninstrumented code (again due to the 4x tracing slowdown). To the extent that they are a consequence of the workload being traced, there are not more interrupts per million instructions executed—each one just occurs relatively sooner in the i-stream. Running an interrupt routine sooner than it would have run in the uninstrumented code can have a subtle effect on the memory access patterns of a processor, but there appears to be no interesting distortion when aggregated over several seconds.

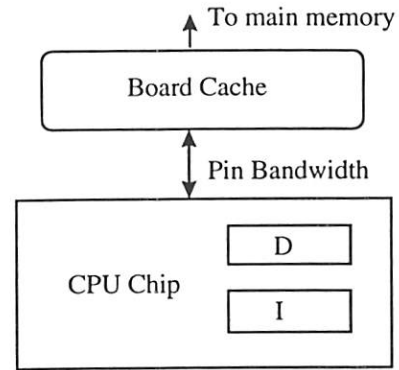


Figure 2: Processor pin bandwidth

3 Pin Bandwidth Study

Our first study using PatchWrx traces is a comparison of processor pin bandwidth requirements for a few different applications running under Windows NT on two Alpha processors. As shown in Figure 2, we are interested in the bandwidth between the processor chip (with on-chip i- and d-caches) and the board-level cache. This memory traffic is due to i- and d-cache misses. The question is: how many bytes per second need to cross the pins of a processor chip when running a given program at full speed? The answer is a function of the workload, including user and system activity, the properties of the processor chip, the clock speed, and the desired clock cycles per instruction (CPI). For our study, “full speed” is 1 CPI. The answers are interesting because, as we’ll see, processor pin bandwidth is a first-order bottleneck for important applications.

3.1 Configuration

We looked at four applications:

1. Microsoft SQL Server, October 1994 beta version, a commercial database server, running the TPC-B [Gra91] benchmark. The trace contains 64.5 million instructions, spanning several hundred transactions.
2. GEM compiler [B⁺92] back-end, from Digital’s commercially available optimizing C compiler, compiling a 3000 line C program. The trace contains 29 million instructions.
3. tomcatv, from the SPEC92 floating point benchmarks [SPE]. The trace contains 47 million instructions.

Application	#instr.	System Ld/st patched	Appl. Ld/st patched	Synthetic data addr.
SQL Server	64M	n	n	73%
GEM comp.	29M	n	Y	8%
Tomcatv	47M	n	Y	24%
Ear	83M	n	n	14%

Table 2: Trace characteristics.

4. ear, also from the SPEC92 benchmarks. The trace contains 83.84 million instructions.

We chose these applications because they give a variety of data points, from heavy memory system usage to light usage. At the time we started the experiments, the SPEC95 benchmarks were not yet available. It would be interesting to do the same experiments for the gcc SPEC95 benchmark, to see how it compares to the GEM results.

We took traces of these applications running on a uniprocessor Windows NT 3.5 system. The traces varied as to how much load/store patching was done. The reasons for not just doing full load/store patching relate to current PatchWrx limitations on image sizes and the slowdowns introduced by load/store patching. As shown in Table 2, none of the uniprocessor traces had loads and stores patched in the operating system. For the SQL Server and ear uniprocessor traces no loads or stores were patched in the application images. For the GEM compiler and tomcatv traces, load and store instructions were patched in the application images. None of the i-stream references in these traces are synthetic, but the d-stream synthetic references vary from 8% to 73%.

We used these traces to simulate the caching behavior of different processor chips under the assumption that the trace executes at 1 CPI. The simulation computes the number of bytes that would need to cross the pins to the board-level cache. These numbers were broken down by the source of the traffic: instruction reference misses, and data reference misses of a variety of kinds.

We looked at the behavior of each application on two different Alpha processors: the Alpha 21064 at 200 MHz, and the Alpha 21164 at 400 MHz.

The Alpha 21064 runs at 200 MHz and has 8 kilobytes each of direct-mapped write-through i-cache and d-cache with a 32-byte cache line size. Each instruction cache miss and data read (load) miss causes 32 bytes to cross the pins. Writes go into a group of four write buffers. A write to a different address causes 32 bytes to cross the pins to flush one of the dirty write buffers. The actual pin bandwidth on this processor is realistically about 300 MB/sec. The maximum bandwidth is about 600 MB/sec.

The Alpha 21164 runs at 400 MHz and has a single 96-kilobyte 3-way associative write-back combined i-cache and d-cache with a 64-byte cache line size. Each i-cache miss and load miss causes 64 bytes to cross the pins. This processor has a write-back cache: writes that hit in the cache do not cause any bytes to cross the pins but do make a cache line dirty. Writes and reads that cause a dirty cache line to be flushed (“victim writes”) cause 16–64 bytes to cross the pins, depending on what portion of the line has been modified; dirty bits are kept for each 16-byte subblock. Writes that miss also cause a cache line to be allocated and filled from memory, so they cause 64 bytes to cross the pins for the read. Note that some of these reads will be unnecessary overhead if all 64 bytes are eventually overwritten (“redundant reads” as compared to “useful reads”). The realistic pin bandwidth on this machine is about 750 MB/sec, and the maximum bandwidth is about 1.6 GB/sec.

3.2 Results

For each of the applications on each of the processors we looked at a plot of the processor pin bandwidth requirements over time. This shows the dynamics of the traffic in addition to giving us upper bounds on the requirements.

3.2.1 SQL Server

Figure 3 shows the simulation results for pin bandwidth requirements on the Alpha 21064 and 21164 processors for the SQL Server workload. The x-axis marks time in chunks of 0.5 million instructions, and the y-axis shows the number of gigabytes per second of traffic across the pins for a given instruction trace.²

The left-hand graph shows the 21064 results. The traffic is broken down into i-cache misses (imiss), d-cache read misses (dmiss), and write buffer flushes (dwrite). At 1 CPI, the pins are overcommitted by a factor of two just on instruction cache misses alone. Three bytes of instruction miss in the cache for every four bytes executed. The required bandwidth for all traffic is about 1.2 GB/sec, or about four times the bandwidth that the processor realistically can deliver. In fact, the uninstrumented workload achieves about 4.3 CPI, giving us confidence that the cache simulations are valid. The simulation is intended to give lower bounds on CPIs based on pin bandwidth requirements; the extra 0.3 CPI in practice is due to non-zero latencies that are not included in the simulation.

²Note: the bandwidth pictures show only the first 25 million instructions of each trace; very little changes in any of the traces beyond that.

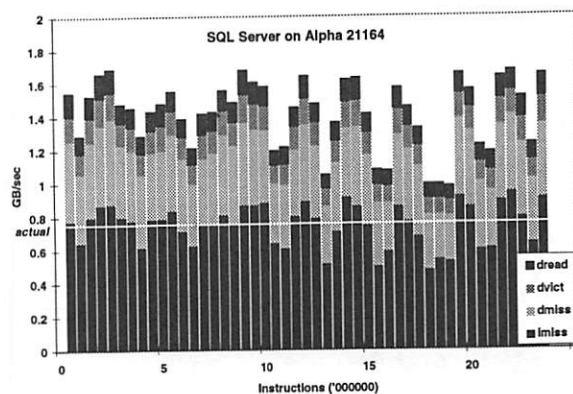
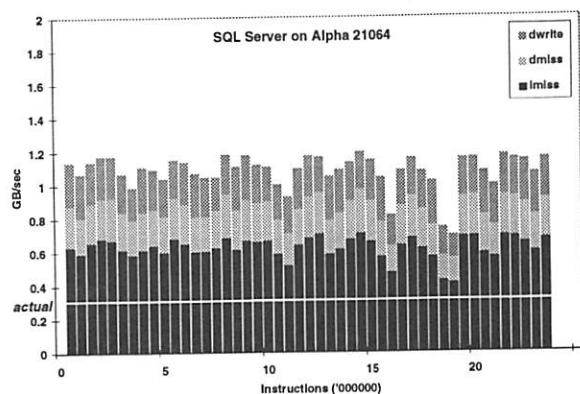


Figure 3: Pin bandwidth requirements of SQL Server running on Alpha 21064 and 21164 processors.

The right-hand graph shows the 21164 results. Here we split up the d-cache write traffic into victim writes (dvict), and useful and redundant reads triggered by write misses (dread). The required pin bandwidth here is as high as 1.7 GB/sec, so the pins are overcommitted by more than a factor of 2 at 1 CPI. The pins are still often overcommitted just on i-cache misses alone, although the i-cache miss traffic periodically dips down below the 750 MB/sec that the processor typically delivers.

3.2.2 Compiler Back-end

The second trace for which we simulated the pin traffic is from the GEM compiler run. The results are shown in Figure 4. The left-hand graph shows that the required pin bandwidth on the Alpha 21064 is about 600 MB/sec. This is about half as much as the SQL Server workload on this machine, with a somewhat lower ratio of i-cache to d-cache traffic. The pins are still overcommitted by a factor of two according to the simulation. The uninstrumented workload on this machine runs at about 2.5 CPI.

The right-hand graph shows the results for this workload on the Alpha 21164 400 MHz processor. Note that the i-stream here mostly fits into the 96-kilobyte cache, with about 200 MB per second of traffic left over. The total pin bandwidth requirements are generally under about 500 MB/sec, with occasional spikes over 800 MB/sec. Thus, pin bandwidth is mostly not a bottleneck for GEM on the 21164.

3.2.3 SPEC Benchmarks

For comparison with the commercial workloads, we chose two of the SPEC92 floating point benchmarks to examine.

Figure 5 shows the simulation results for the tomcatv trace. On the 21064, the i-stream essentially fits into the 8 kilobyte i-cache. The program repeatedly goes through

phases where the pins are overcommitted by factors of 1.5 to 3 or more, and then are okay. The actual program runs at between 1 and 1.5 CPI on this machine. On the 21164, more of the workload falls within the bandwidth limitations, with the periodic lower plateaus well within the capacity of the processor pins, but there are still periodic benchmark array access bursts that are factors of two or more higher than the available bandwidth. Note that the instruction stream mostly fits in the cache, and so doesn't contribute much to the bandwidth requirements. Also, the demands for data bandwidth fall into a predictable pattern. Perhaps prefetching during times when bandwidth is underutilized could help for this workload.

Figure 6 shows the simulation results for the ear trace running on the same two processors. We see that even on the 21064, the pin bandwidth is essentially sufficient with required bandwidth about 200 MB/sec. The actual workload runs at 1 CPI or less (because of the dual instruction issue capabilities of the processor). The requirements for ear on the 21164 400 MHz processor are well within the range of the processor. A large part of the workload fits in the caches in this case.

3.3 Memory Maps

Looking at the patterns of memory accesses over time helps illuminate the results of the pin bandwidth simulations shown above. Figures 7–11 show the memory footprints for the first 25 million instructions in each trace³. The x-axis shows time from left to right, in millions of instructions executed. The y-axis shows virtual addresses accessed, modulo 4 MB (chosen because it is an interesting board cache size). There is a dot in the picture for each instruction address and for each load/store data address.

³[except for the ear trace, which has about 21 million instructions shown in this version]

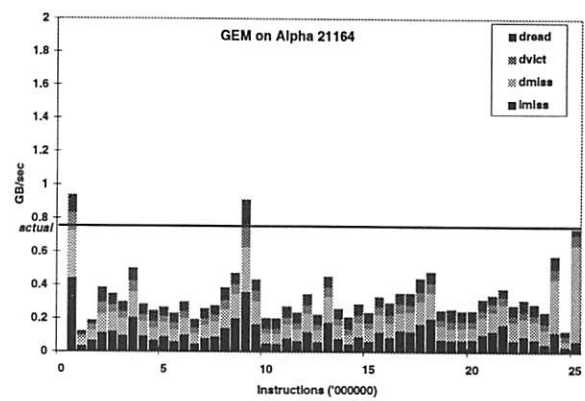
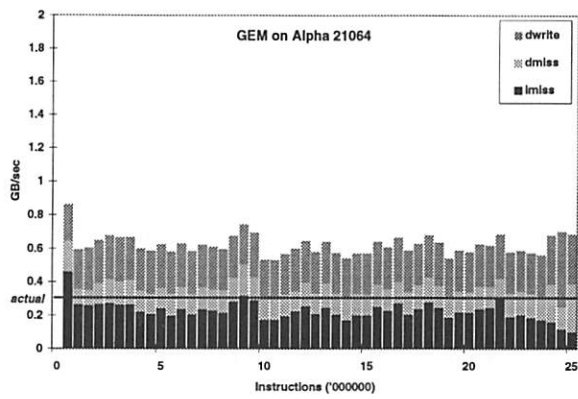


Figure 4: Pin bandwidth requirements of GEM compiler running on Alpha 21064 and 21164 processors.

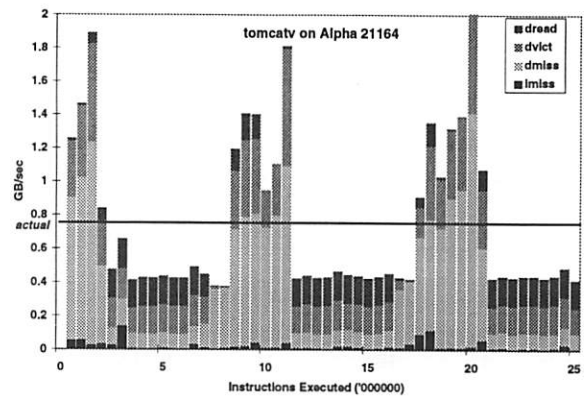
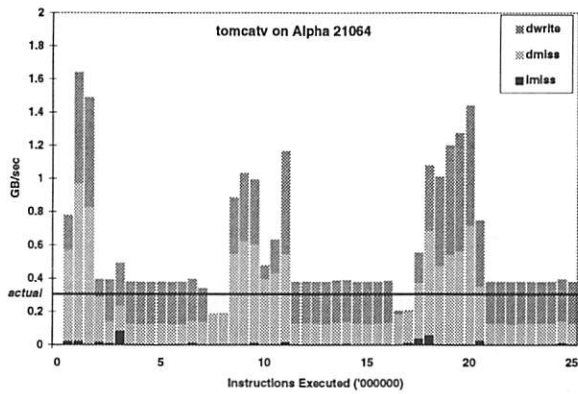
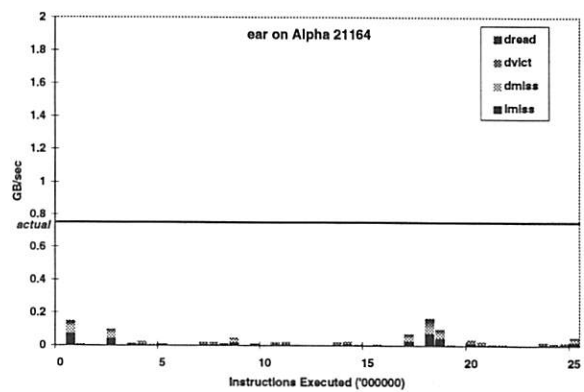
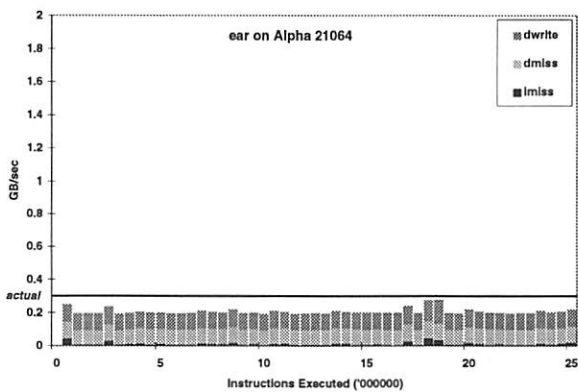


Figure 5: Pin bandwidth requirements of tomcatv running on Alpha 21064 and 21164 processors.



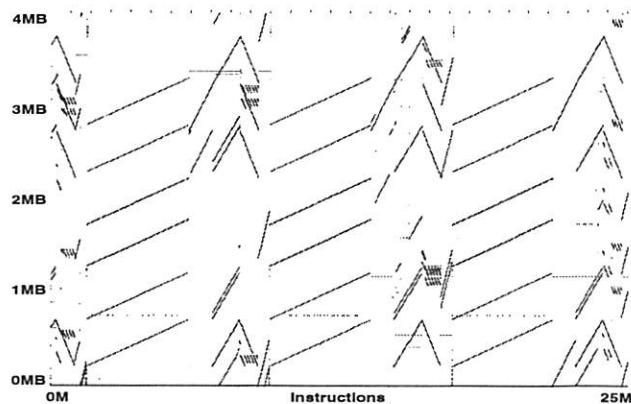


Figure 7: tomcatv memory access plot showing the distribution of user-space addresses over time. Looking at any vertical slice we see all the addresses (mod 4 MB) that were accessed within that time slice.

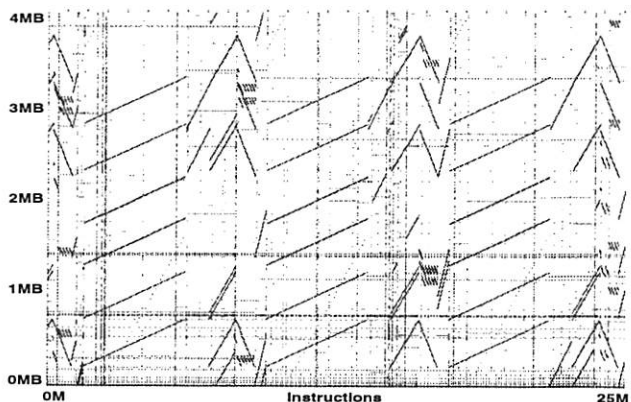


Figure 8: tomcatv memory accesses, user and system.

Figure 7 shows the memory footprint for the user-only part of the tomcatv benchmark. The bottom solid line shows the instruction fetches for the benchmark, all in a tight loop on a single page of code. The leftmost group of six upward-sloping lines are data references sweeping forward through six half-megabyte arrays. These are followed by some faster forward sweeps through different arrays, four backward sweeps, and the data pattern repeats. This program has very predictable branching patterns and very predictable data accesses.

In Figure 8, we have added the operating system part of the trace. The periodic vertical ticks about one sixth of the way up from the bottom of the picture are the i-stream for the timer interrupt routine (about four times too often because of the tracing slowdown). The other dots that correlate with these are the data references made in the timer routine. The occasional vertical lines of high-density references are other operating-system activity, including the thread scheduler and network traffic. This operating system code has less predictable branch-

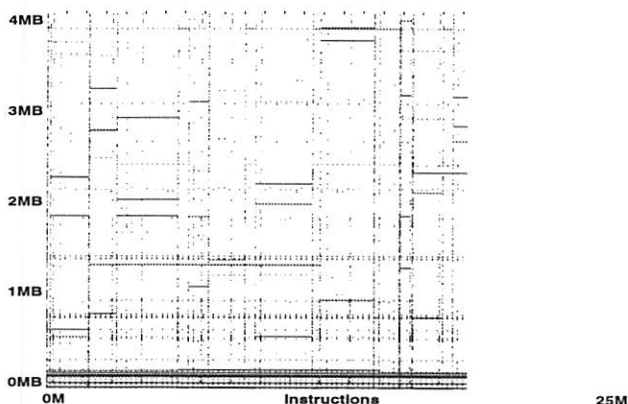


Figure 9: ear memory accesses, user and system.

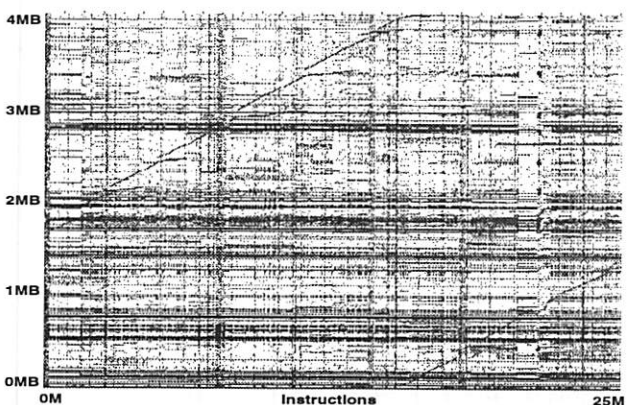


Figure 10: GEM memory accesses, user and system.

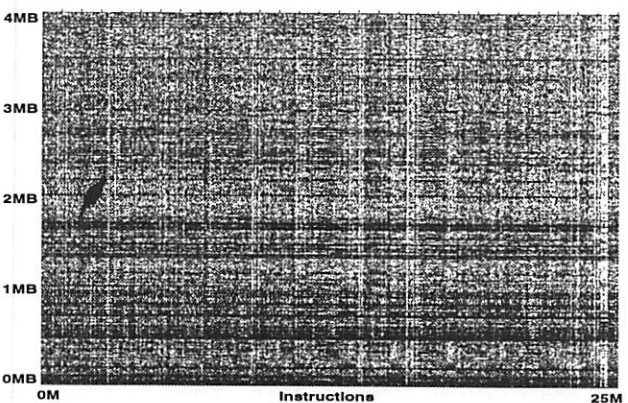


Figure 11: SQL Server memory accesses, user and system.

ing patterns and data accesses.

Figure 9 shows, on the same scale, a trace of both system and user addresses for the ear benchmark. The memory usage is relatively light, with few strong patterns other than the horizontal lines due to timer interrupts and some accesses to application data structures, and the vertical lines due to operating system activity.

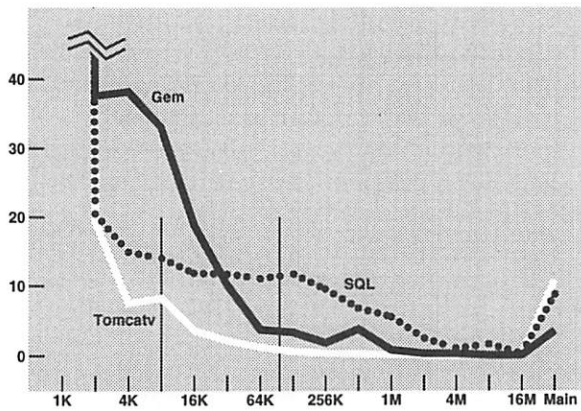


Figure 12: Incremental I-cache Hit Curves.

Figure 10 shows, on the same scale, a trace of the GEM compiler benchmark. The horizontal lines represent frequently used subroutines and frequently used data. The visually dominant diagonal line appears to be the buffer of program text being compiled; the compiler is sweeping forward through it (modulo 4 MB). The timer-interrupts are still there, but a bit harder to see. Most of the rest of the dots are data references to linked-list data structures used by the operating system. This workload has less predictable memory access patterns than tomcatv.

Finally, Figure 11 shows, on the same scale, a trace of the SQL Server benchmark. The trace looks somewhat similar to white noise—there is very little correlation between addresses. This application falsifies the premise of a cache: that accessing a location is a good predictor that the same or a nearby location will be accessed in the near future.

These memory maps highlight the differences between the memory requirements of the workloads. There seems to be at least some rough correlation between the density of the memory maps and the pin bandwidth requirements.

3.4 Cache Demand in More Detail

To understand the processor bandwidth results better, we looked at the *incremental cache hit* curves for cache sizes from 1KB to 16MB. Except for the smallest cache, the incremental hit rate for a given size cache is the number of *additional* hits that occur in the given cache and miss in smaller caches. Portions of the incremental hit curve for sizes bigger than processor on-chip caches represent misses that produce the off-chip bandwidth demand.

In this section, we used direct-mapped combined i- and d-stream caches and a line size of 64 bytes, and ran the simulations over the initial 25M instructions of each trace.

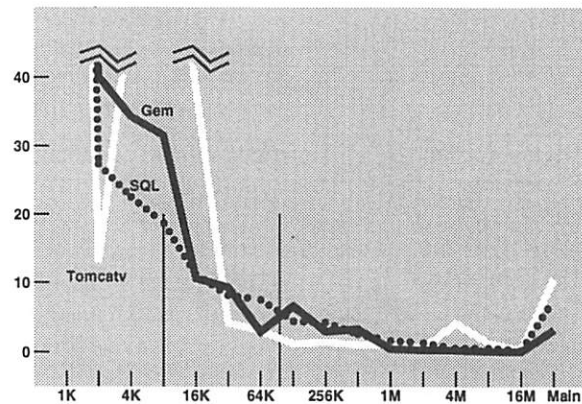


Figure 13: Incremental D-cache Hit Curves.

From the bandwidth charts above, we see a marked difference in i-stream miss rates across the uniprocessor workloads. Figure 12 shows the incremental i-cache hit curves for three of them (we exclude ear from this Section's discussion because its memory demands are uninteresting). For all three curves, the 1KB points are off the chart at 845-957 hits/1000 instructions. This is to be expected – if straightline code is executed with no repetition from 64-byte (16 instruction) cache lines, we would expect 15/16 hits, or 937 hits/1000 instructions. The short vertical bars mark 8K and 96K cache sizes.

The tomcatv i-stream fits almost entirely in 8KB, with a little tail that hits only in main memory. The GEM compiler i-stream extends out to 32KB before it drops off, and the SQL i-stream has significant incremental hits all the way out to a 1MB cache. This large footprint is why the SQL i-stream puts such a bandwidth load on the processor pins. This result is roughly consistent with the i-cache miss rates measured by [MDO94] for TPC-A and TPC-C (although they had a larger percentage of time in operating system code than we observed). The GEM footprint produces a large bandwidth load with an 8KB on-chip cache but a small load with a 96KB cache. The tomcatv footprint presents very little off-chip bandwidth load.

Figure 13 shows the incremental d-cache hit curves. All three d-streams show a tail of references that miss even in 16MB and go all the way to main memory. The tomcatv curve has peaks at 8KB (76 misses per 1000 instructions) and 4MB that correspond to sizes of frequently accessed arrays in that program. The 4MB peak is why the tomcatv d-stream puts a large periodic bandwidth load on the processor pins.

In Figure 14, we have broken down the SQL incremental i-cache hits by system and user code. The system hits are per 1000 system instructions, and the user hits per 1000 user instructions. The SQL system code is about 25% of the total i-stream. The system code not only has

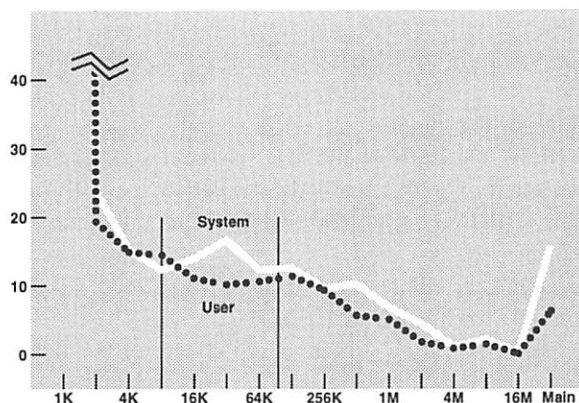


Figure 14: SQL User vs. System I-stream.

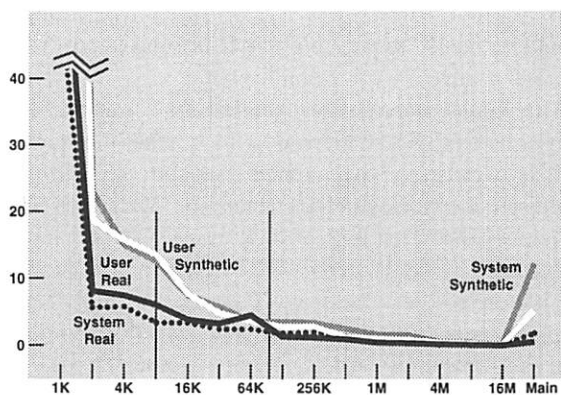


Figure 15: SQL User vs. System D-stream.

a bigger cache footprint, it has a larger tail of hits in main memory.

The SQL d-stream breakdown, Figure 15, shows system vs. user and also real vs. synthetic d-stream addresses. The d-stream has a high proportion of synthetic addresses, so is not very precise. The major difference between the real and synthetic addresses is a larger proportion of synthetic addresses that miss all the way out to main memory. For both the real and synthetic d-stream addresses, the system code has a larger footprint than the user code.

Both the GEM compiler and tomcatv i-stream breakdowns (not shown here) reveal the system i-stream to have a much larger footprint than the user i-stream. Similarly, the d-stream breakdowns show the system d-stream to have a much larger footprint than the user one (although tomcatv is close, due to the large array accesses).

In addition to observing larger footprints in system and SQL server code, the incremental cache hit curve allows us to compute the relationship between the CPI of a workload and the performance of various levels of the memory hierarchy. If we multiply each point of the incremental cache hit curves by the time it takes to service

	21064	21164
SQL	3.13	1.87
GEM	1.20	0.28
tom	0.04	0.06
ear	0.05	0.02

Table 3: Average bytes per instruction of pin traffic due to the i-streams of each of the benchmarks on the two processors studied.

that hit in a real memory subsystem, we get the total time spent waiting on memory accesses for that workload.

Using realistic times for early Alpha computer systems, the SQL i-stream and d-stream hits beyond 96KB each total about 1.8 CPI. Combined with a 100%-hits execution rate of about 1 CPI, these total about 4.6 CPI. This is roughly consistent with the measurements in [CB94] of 4.3 CPI, and 20-25% operating system time, and 39%+36% i-stream plus d-stream stall time, on the TP1 database workload on an older AlphaSystem 7000 computer.

3.5 Discussion

We conclude from these experiments that processor pin bandwidth is a bottleneck for some applications.

SQL Server needs about two to four times the bandwidth of tomcatv, and is limited to about 4 CPI on at least one current processor. According to another experiment that we ran, a future processor similar to the Alpha 21164 but running at 500 MHz would need at least 2 GB/sec of pin bandwidth to run the SQL Server benchmark at 1 CPI.

Both SQL Server and GEM have higher pin bandwidth requirements due to i-stream traffic than either of the SPEC benchmarks, as summarized in Table 3. This i-stream traffic contributes significantly to the pin bandwidth bottleneck for SQL Server. Therefore, chip designers should be looking at more than SPEC benchmarks if they want their future chips to run commercial workloads well.

Our experiments also suggest that it is increasingly important for algorithm designers to pay attention to memory structure and cache parameters if they want their code to run fast. One example of such an effort is the AlphaSort work [NBC⁺94], where careful attention to these memory details paid off handsomely.

4 Multiprocessor Study

Our second study examines the locking behavior of SQL Server running the TPC-B benchmark on a four-processor AlphaServer 2100 symmetric multiprocessor.

The traces were taken for SQL Server version 6.00.85a (March 1995 beta release), running Windows NT 3.5, with 12 clients running transactions against the database. We chose this number of clients to minimize the idle time of the system, which ended up being about 2–3%.

The trace contains about 69 million instructions. In contrast to the uniprocessor traces, operating system loads and stores are patched. For the application code, only loads and stores to lock variables are patched.

We use the trace in two ways. First, we look at a picture showing the timeline of software lock acquisition and release activity. Then, we simulate the cache coherence protocol employed on the machine to understand how the locking activity affects communication among the processors.

4.1 Software Lock Activity

Figure 16 shows the timeline of locking activity for a 20-millisecond portion of the SQL Server trace, encompassing portions of several transactions on the four processors. The picture shows groups of four lines (with backgrounds alternately shaded white and light gray), one line per processor, with shaded bars representing the elapsed times that locks are held. Time runs from left to right and top to bottom. Each group of four horizontal lines represents 2 milliseconds of elapsed time.

In this picture, we distinguish three cases: the thick black bars are the operating system's dispatcher lock (KiDispatcherLock), the thick gray bars are all other lock holding times, and the thin black lines are times spent spinning while waiting to acquire a lock.⁴ In the rectangle at 0.030 seconds, for example, processor 0 starts out holding no locks, processor 1 holds the dispatcher lock, and processor 2 is spinning on the dispatcher lock, and processor 3 holds no locks. Processor 1 then releases the dispatcher lock and processor 2 acquires it. Four other locks are used, then processors 1 and 2 use the dispatcher lock again. Finally, processor 1 uses another lock and processor 0 uses a lock.

Lock acquisitions and releases are not reported directly in the trace, but rather, must be detected by pattern matching against the common sequences of instructions used for this purpose. Therefore, the picture showing lock acquisition and release points is not perfect. However, we have taken some care to ensure that the picture is reasonably self-consistent, and consistent with the code.

While this picture shows just a small slice of the trace, it reveals several interesting behaviors. The dispatcher lock is held for relatively long stretches of time (200–900 instructions). Also, it is held for a large percentage of

⁴We usually look at this picture in color, with different colors and line widths for different locks, making more of the locking behavior apparent at once.

the time—about 45% of the total elapsed time. This suggests that the workload could not scale up beyond eight processors, which would cause this lock to be held nearly 100% of the time.

More than 16% of the time is spent spinning while waiting for a lock, usually the dispatcher lock. This spin time would go up substantially as the lock-held time approaches 100%. Note the convoy effect in the second rectangle, at about 0.041 seconds. Processors line up spinning for the dispatcher lock.

During the long stretch where processor 0 holds the dispatcher lock, a disk interrupt arrives and the processor services it while still holding the lock. Other processors are waiting to get the lock during that time.

All of the evidence above suggests that the dispatcher lock is a bottleneck. Because of the convoy effect, it would be nearly impossible to get beyond about 6 processors' worth of work from this code.

The picture reveals that in addition to being held for a long time overall, the dispatcher lock is held frequently. This is because the SQL Server user code goes through the dispatcher lock to block when one of its own locks is unavailable. This type of lock usage was perhaps not anticipated by the Windows NT implementers. Showing the picture to the Windows NT kernel and SQL Server implementers at the same time brought out assumptions that each group was making about the other's code.

The full-color version of this picture makes other locking patterns visible. For example, there is a repeated pattern of nine lock acquisitions (some of which are the same lock repeatedly acquired and released) in the same order when manipulating the operating system page tables. Examining the code may suggest ways to make the associated operation more efficient.

4.2 Interference from Cache Coherence

The software lock activity above for achieving mutual exclusion is best tuned by changing the software. We also looked at multiprocessor hardware cache interference caused by writes on one processor affecting caches on other processors.

We looked in detail at the behavior of the cache coherence protocol with four AlphaServer 2100 processors. A pure invalidate protocol is used—a write by one processor to a cache line invalidates all other cached copies of the line in other processors. We found that 40% of the board-level cache misses in the 4-processor SQL trace were due to interference from other processors—invalidate due to writes of shared variables.

Most of these misses were in fact to about 24 cache lines containing software locks. (Note that the time lost on servicing these misses is smaller than the time lost above spinning on held locks.) Some of the invalidations

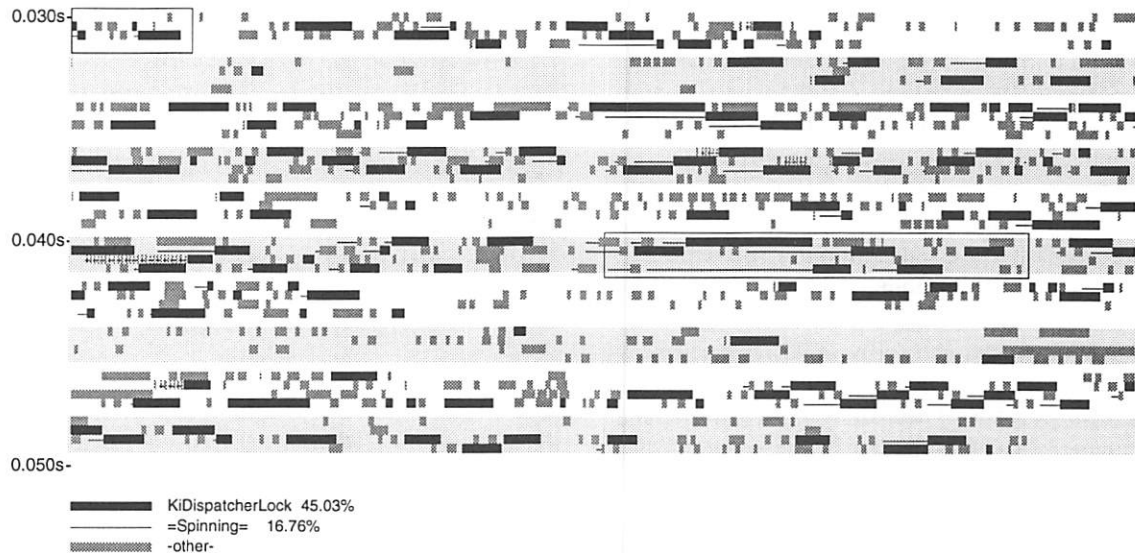


Figure 16: SQL Server lock activity. Time runs left-to-right and top-to-bottom.

came from false sharing. The programmers who wrote the code assumed 32-byte cache lines and took care to put locks in different 32-byte lines than other variables. However, this particular machine has 64-byte lines, so a lock and another unrelated shared variable could end up in the same line and cause unnecessary invalidations. The ability to track shared sub-blocks in the hardware would help this problem. A better cache coherence protocol for this workload would be a hardware design that writes through for shared blocks, having recipients update or invalidate depending on how recently they had touched the line. Such a design was used in the Alpha Demonstration Unit [TCS93].

4.3 Discussion

The Windows NT 3.5 operating system code holds the dispatcher lock for so long because it is doing a full context switch while holding the lock. It would be better to make the decision about what to dispatch next while holding the lock, release the lock, then perform the actual context switch without delaying the other processors. Allowing interrupts while holding the dispatcher lock also lengthens the path and slows down other processors. It would be better to turn off interrupts before acquiring the lock and turn them back on after releasing the lock.

The application code that triggers too-frequent use of the dispatcher lock could adopt other locking strategies, such as trying a handful of times to acquire an application lock before blocking or to do user-level dispatching between multiple transactions.

Examining the dynamic locking behavior of the sys-

tem reveals properties that are more difficult or impossible to see from statistical information. We can see clearly the convoy effect for locks that are in high demand, and we can see patterns of locking that may suggest inefficiencies in the code. We can also identify cases where locks are probably not a performance problem, as is true for most of the locks in our trace.

Having the traces available for simulating cache coherence protocols helped us to understand the causes of scaling problems in our multiprocessor system, and can help the designers of future systems to avoid these problems.

5 Conclusions

These studies have demonstrated that full, dynamic execution traces are useful for designing future systems, as well as for diagnosing difficult performance problems of current systems.

Obtaining such traces is practical using PatchWrx. The overhead is low enough on the uniprocessor systems (about a factor of 2 slowdown) that we always ran the systems with instrumentation in place. Overhead on multiprocessor systems was reasonable enough to get useful traces. PatchWrx traces are in the range of about 50–100 million instructions, which was enough to observe interesting dynamic behavior in the system while still being practical to collect.

The specific versions of software that we studied are inevitably already out of date with respect to currently available versions, and some of the specific problems revealed in our studies may have been fixed by now. In fact,

the Microsoft SQL Server 95 and Windows NT 3.51 and 4.0 products contain performance improvements⁵ that resulted from our studies [Gra95]. However, the same techniques and tools can be used to look for performance problems in the new versions and the traces still provide valuable input for testing future chip designs.

We find that real pieces of large software are bigger than any practical on-chip caches, and therefore use much more processor pin bandwidth. Pin bandwidth can be a bottleneck to running commercial workloads at the desired speed. For the near future at least, computer design is memory design, and software design is memory design. Pin bandwidth is an important consideration (as much as memory latency) because it sets a lower bound on the number of clock cycles per instruction needed to execute a workload.

Although this paper emphasizes the effect of bandwidth on the performance of these processors, one shouldn't ignore the effect of latency. If there is insufficient bandwidth to perform a given workload, the workload will run proportionally slowly, and there is nothing software can do to compensate for this. Systems that have sufficient bandwidth but high memory-access latency can possibly perform well, but only for software that is designed to hide the high latency. Both high bandwidth and low latency are necessary for peak performance on most commercial software.

6 Acknowledgments

A number of people at Digital have contributed to this effort. Benn Schreiber, Tom van Baak, Miche Baker-Harvey, and Joe Notarangelo gave us immeasurable help in getting the NT PAL-code modifications running. Rich Witek wrote the original single-processor boot PAL-code. Mike Burrows contributed expertise on binary modification techniques. Wim Colgate provided the key insight into booting NT multiprocessor systems. Dave Hunter generously provided time on a four-processor AlphaServer 2100 system. Chuck Thacker, Dave Conroy, Bill Weihl, Greg Nelson, Amitabh Srivastava and Alan Eustace provided continuing encouragement. Our summer interns, Cliff Mercer and David Martin struggled with early software and each provided key insights for increasing performance of the workloads studied. David Martin was responsible for the cache coherence protocol

⁵Improvements include: some collapsing of the 36 memory allocation calls per TPC-B transaction, each of which calls EnterCriticalSection; replacing a slow locking sequence macro expansion with a more efficient one, worth 5-8% all by itself; removing unnecessary calls to floating point initialization routines; avoiding calls to OS services for CPU time and clock time; compiler improvements in setjmp/longjmp implementation; minimizing context switches; and increasing cache locality.

simulations. We also thank Jim Gray and David Cutler at Microsoft.

The observations of Richard Draves and the anonymous OSDI referees substantially improved the paper.

References

- [AH90] Anant Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. *Performance Evaluation Review*, 18(1):48-57, May 1990.
- [ASH86] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proc. of the 13th International Symposium on Computer Architecture*, pages 119-127, June 1986.
- [B⁺92] David S. Blickstein et al. The GEM optimizing compiler system. *Digital Technical Journal*, 4(4), 1992. Also available as <http://ftp.digital.com/pub/Digital/DECinfo/DTJ/axp-gem.ps>.
- [BKW90] Anita K. Borg, R. E. Kessler, and David W. Wall. Generation and analysis of very long address traces. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 270-279, May 1990.
- [CB93] Brad J. Chen and B. Bershad. The impact of operating system structure on memory system performance. In *Proc. of the 16th International Symposium on Operating Systems Principles*, pages 120-133, December 1993.
- [CB94] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP performance using TP and SPEC workloads. In *Proc. of the 21st Annual Symposium on Computer Architecture*, pages 60-70, April 1994.
- [CHRG95] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *ACM SIGMETRICS*, pages 1-13, May 1995.
- [CK94] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS*, pages 128-137, May 1994.
- [EKKL90] Susan J. Eggers, David Keppel, Eric J. Kolding, and Henry M. Levy. Techniques

- for efficient inline tracing on a shared-memory multiprocessor. In *ACM SIGMETRICS*, pages 37–47, May 1990.
- [Gra91] Jim Gray, editor. *The Benchmark Handbook*, chapter 2, pages 79–117. Morgan Kaufmann, San Mateo, California, 1991.
- [Gra95] Jim Gray, December 1995. Private communication.
- [LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.
- [MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, October 1994.
- [NBC⁺94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC machine sort. In *SIGMOD Conference*, pages 233–242, 1994.
- [SCK⁺93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
- [SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 1–1, 1994.
- [SJF92] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing of parallel systems via TRAPEDS. *Microprocessors and Microsystems*, 16(5):249–261, 1992.
- [SPE] SPEC CPU92 benchmarks. World Wide Web URL: <http://www.specbench.org/osg/cpu92/>.
- [SW95] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, Newton MA, 2nd edition edition, 1995.
- [TCS93] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart. The Alpha Demonstration Unit: A high-performance multiprocessor. *Communications of the ACM*, 36(2):55–67, February 1993.
- [TGH92] J. Torrellas, A. Gupta, and John Hennessy. Characterizing the cache performance and synchronization behavior of a multiprocessor operating system. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

Using Latency to Evaluate Interactive System Performance

Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer
Harvard University, Division of Engineering and Applied Sciences
{yaz, zhwang, bchen, margo}@eecs.harvard.edu

Abstract

The conventional methodology for system performance measurement, which relies primarily on throughput-sensitive benchmarks and throughput metrics, has major limitations when analyzing the behavior and performance of interactive workloads. The increasingly interactive character of personal computing demands new ways of measuring and analyzing system performance. In this paper, we present a combination of measurement techniques and benchmark methodologies that address these problems. We introduce several simple methods for making direct and precise measurements of event handling latency in the context of a realistic interactive application. We analyze how results from such measurements can be used to understand the detailed behavior of latency-critical events. We demonstrate our techniques in an analysis of the performance of two releases of Windows NT and Windows 95. Our experience indicates that latency can be measured for a class of interactive workloads, providing a substantial improvement in the accuracy and detail of performance information over measurements based strictly on throughput.

1 Introduction

Benchmarks are used in computer systems research to analyze design alternatives, identify performance problems, and motivate improvements in system design. Equally important, consumers use benchmarks to evaluate and compare computer systems. Current benchmarks typically report throughput, bandwidth, or end-to-end latency metrics. Though often successful in rating the throughput of transaction processing systems and/or the performance of a system for scientific computation, these benchmarks do not give a direct indication of performance that is relevant for interactive applications such as those that dominate modern desktop computing. The most important performance criterion for interactive applications is responsiveness, which determines the performance perceived by the user.

In this paper, we propose a set of new techniques for performance measurement in which latency is measured in the context of a workload that is realistic, both in

terms of the application used and the rate at which user-initiated events are generated. We present low-overhead methods that require minimal modifications to the system for measuring latency for a broad class of interactive events. We use a collection of simple benchmark examples to characterize our measurement methodology. Finally, we demonstrate the utility of our metrics by applying them in a comparison of Microsoft Windows NT versions 3.51 and 4.0 and Windows 95, using realistic interactive input to off-the-shelf applications.

The remainder of this section provides background on the problem of measuring latency, including the motivation for our new methodology based on an analysis of the current practice in performance measurement. Section 2 describes our methodology in detail. In Section 3, we discuss some of the issues in evaluating response time in terms of a user's experience. In Sections 4 and 5, we apply our methodology in a comparison of Windows NT versions 3.51 and 4.0 and Windows 95. Sections 6 and 7 discuss the limitations of our work and conclude.

1.1 The Irrelevance of Throughput

Most macrobenchmarks designed for interactive systems use throughput as the performance metric, measuring the time that the system takes to complete a sequence of user requests. A key feature of throughput as a performance metric is that it can be measured easily, given an accurate timer and a computation that will do a fixed amount of work. Throughput metrics measure system performance for repetitive, synchronous sequences of requests. However, the results of these benchmarks do not correlate directly with user-perceived performance—a critical metric when evaluating interactive system performance. The performance of many modern applications depends on the speed at which the system can respond to an asynchronous stream of independent and diverse events that result from interactive user input or network packet arrival; we call this *event handling latency*. Throughput metrics are ill-equipped to characterize systems in such ways. More specifically, throughput benchmarks fail to provide enough information for evaluating interactive system performance and make inappropriate assumptions for measuring interactive systems.

This work was sponsored in part by grants from Bellcore, Intel, the National Science Foundation, Sun Microsystems Laboratories, and the Sloan Foundation.

Information Lost

The results of throughput benchmarks are often reduced to a single number that indicates how long a system took to complete a sequence of events. Although this can provide information about the sum of the latencies for a sequence of events, it does not provide information about the variance in response time, which is an important factor in determining perceived interactive performance.

The insufficient detail provided by throughput benchmarks can also mislead designers trying to identify the bottlenecks of a system. Since throughput benchmarks provide only end-to-end measures of activity, system activity generated by low-latency events cannot be distinguished from that generated by longer-latency events, which have a much greater impact on user-perceived performance. Worse, if such a benchmark includes sufficiently many short-latency events, these short events can contribute significantly to elapsed time, leading designers to optimize parts of the system that have little or no impact on user-perceived performance. In an effort to compare favorably against other systems in throughput benchmarks, designers may even undertake such optimizations knowingly. In this case, bad benchmarking methodology hurts both system designers and end users.

In addition, user interfaces tend to use features such as blinking cursors and interactive spelling checkers that have (or are intended to have) negligible impact on perceived interactive performance, yet may be responsible for a significant amount of the computation in the overall activity of an application. Throughput measures provide no way to distinguish between these features and events that are less frequent but have a significant impact on user-perceived performance.

Inaccurate User Assumptions

Throughput benchmarks often drive the system by feeding user input as rapidly as the system can accept it—equivalent to modeling an infinitely fast user. Such an input stream is unrealistic and susceptible to generating misleading results. One of the sources for such errors is batching. Client-server systems such as Windows NT and the X-Window system batch multiple client requests into a single message before sending them to the server. This reduces communication overhead and allows the server to apply optimizations to the request stream, such as removing operations that are overridden by later requests. Although batching improves throughput, it can have a negative effect on the responsiveness of the system.

When a benchmark uses an uninterrupted stream of

requests, the system batches requests more aggressively to improve throughput. Measurement results obtained while the system is operating in this mode are meaningless; users will never be able to generate such an input stream and achieve a similar level of batching in actual use. Disabling batching altogether is sometimes possible but does not fully address the problem. An ideal test input should permit a level of batching that is likely to occur in response to real user input.

Overall, throughput measures provide an indirect rather than a direct measure of latency, and as such they can give a distorted view of interactive performance. An ideal benchmarking methodology will drive the system in the same way that real users do and give designers a correct indication as to which parts of the system are responsible for delays or user-perceptible latency. Obtaining such figures requires that we drive the system using an input stream that closely resembles one that an interactive user may generate and more importantly, an ability to measure the latency of individual events.

1.2 Related Work

Most of the benchmarks currently used for computer performance measurement can be categorized as either microbenchmarks or batch benchmarks. Microbenchmarks are commonly used to measure the performance of relatively simple, specific events [7,10]. They can be used to measure latency (e.g., 7.7 μ s per IPC) or throughput (e.g., 129,000 local IPCs per second). Some examples of current popular microbenchmark suites are Winbench [14], the Bytemarks [3], and *lmbench* [7]. The shortcomings of microbenchmarks for general performance analysis are well documented [2]. The fundamental problem is that the computation they perform is not useful in and of itself and, as such, they fail to accurately reflect system behavior for any realistic situation.

Batch benchmarks address the shortcomings of microbenchmarks by measuring the time to run a complete, non-trivial computation from start to finish. Batch benchmarks are the dominant way of measuring system performance [1,13,14]. Portable benchmarks such as the SPEC95 suite are based on batch computations with non-interactive input. The portability requirement means that it must be possible to adapt these benchmarks to use new system APIs or to accommodate the idiosyncrasies of a given API implementation. To meet these requirements, the SPEC benchmarks are distributed in source form. The central problem with benchmarks such as the SPEC suites is that, although they may represent a realistic batch load, they fail to model the behavior of an interactive user [8].

Benchmarks such as Winstone, BAPCo SYSmark NT

and BAPCo SYSmark 32 sacrifice portability in order to use popular interactive applications. Winstone is specific to PC-compatible hardware running Windows 95 or Windows NT. SYSmark NT provides somewhat more hardware portability than the Winstone suite, providing native executables for non-x86 systems that run Windows NT. Although these benchmark suites use interactive applications and simulated interactive input, they report performance in terms of throughput metrics and suffer from the problems presented in Section 1.1. Non-trivial batch computations are sometimes included in these workloads when they are consistent with realistic usage of the application. Examples are circuit board layout in MAXEDA (BAPCo SYSmark NT) and database queries in Microsoft Access (Winstone 95). Although these benchmarks use realistic applications, the input streams used to drive them model an infinitely fast user.

BAPCo SYSmark for File Servers, Ziff-Davis Netbench, SPEC SFS (LADDIS), and the TPC benchmarks measure performance of a server under client load. Both Netbench and SPEC SFS measure file server performance, and compute their results using the latency observed by clients. The intent of TPC-C is to measure on-line transaction processing performance. It reports performance in terms of complete transactions processed per minute. TPC-D is designed to measure decision-support database performance. TPC-D specifies one metric (QppD) that measures “raw performance” or latency, and another metric (QthD) for throughput.¹ Several of these benchmarks use client-observed latency in reporting performance, but only for client requests to a remote server. Our goal is to measure latency for a more general class of events.

A prior study compared Windows NT, Windows for Workgroups 3.11, and NetBSD to explore how differences in system structure affect overall performance [4]. The study used microbenchmarks to measure the latency of simple events and application workloads. Although some of the application workloads were based on interactive applications, the applications were driven by an uninterrupted input stream, and the results were reported in terms of throughput. There was no attempt to measure interactive performance directly. Because of the requirement that the workloads run on a Unix system, the study included no popular PC applications.

Two prior studies influenced our work. Shand used a free-running counter in a tight loop to measure the latency of interrupts [11]. This methodology is similar to ours in that it measures the computation of interest by

detecting lost time. We generalize the prior work by looking beyond interrupt handling to a broader range of events types, by avoiding the need for special-purpose hardware, and by looking at interactive events rather than the latency of interrupt handlers. Second, our techniques for visualizing latency were influenced by the work of O’Toole et al. [9] on reducing the pause times for garbage collection.

2 Methodology

Our methodology must provide the ability to measure the latency of individual events that occur while executing realistic interactive workloads. This poses the following set of new challenges:

- Interactive events are short in duration relative to the timer resolution provided by clock APIs in modern operating systems such as Windows and UNIX. Whereas a batch workload might run for millions of timer ticks, many interactive events last less than a single timer interval.
- Under realistic load, there will often be only a fraction of a second between interactive events in which to record results and prepare for the next measurement. Therefore the measurement scheme must have quick turnaround time.
- Perhaps the most challenging problem is collecting the requisite data without access to the source code of the applications or operating system. With source code, it is straightforward to instrument an application to generate timestamps at the beginning and ending points of every interactive event, but this is time consuming at best and not possible given our goal of measuring widely-available commercial software.

Analyzing interactive applications is just as challenging as measuring them. The time during which an application is running can be divided into *think time* and *wait time*. Think time is the time during which the user is neither making requests of the system nor waiting for the system to do something. Wait time is the time during which the system is responding to a request for which the user is waiting. Not all wait time is equivalent with respect to the user; wait time intervals shorter than a user’s perception are irrelevant. We call these classes of wait time “unnoticeable.” A good example of unnoticeable wait time is the time required to service a keystroke when a user is entering text. Although the system may require a few tens of milliseconds to respond to each keystroke, such small “waits” will be unnoticeable, as even the best typists require approximately 120 ms per keystroke [12]. Distinguishing between wait time and

1. TPC-A and TPC-B were declared obsolete by the Transaction Processing Council in June of 1995.

think time is non-trivial, and the quantity and distribution of wait time is what the user perceives as an application's responsiveness. Our measurement methodology must help us recognize the wait time that is likely to irritate users.

In the following sections, we describe the combination of tools and techniques that we use to measure and identify event latency.

2.1 Experimental Systems

We ran our experiments on a personal computer based on an Intel Premiere II motherboard, with the Intel Neptune chip set and a 100 MHz Pentium processor. Our machine was equipped with a 256KB asynchronous SRAM Level 2 cache, 32 MB of RAM, and a Diamond Stealth 64 DRAM display card. We used a dedicated 1GB Fujitsu disk (model M1606SAU) for each of the operating systems we tested. These disks were connected via a NCR825-based SCSI II host adapter. Both Windows NT systems used a NTFS file system, while the Windows 95 system used a FAT file system. The significant differences between Windows NT 3.51 and NT 4.0 are a change in GUI (NT 3.51 uses the traditional Windows GUI, while NT 4.0 uses a GUI similar to that of Windows 95) and the movement of some Win32 components into the kernel.

2.2 The Pentium Counters

The Intel Pentium processor has several built-in hardware counters, including one 64-bit cycle counter and two 40-bit configurable event counters [5]. The counters can be configured to count any one of a number of different hardware events (e.g., TLB misses, interrupts, or segment register loads). The Pentium counters make it possible to obtain accurate counts of a broad range of processor events. Although the cycle counter can be accessed in user or system mode, the two event counters can only be read and configured from system mode.

2.3 Idle Loop Instrumentation

Our first measurement technique uses a simple model of user interaction to measure the duration of interactive events. In an interactive system, the CPU is mostly idle. When an interactive event arrives, the CPU becomes busy and then returns to the idle state when the event-handling is complete. By recording when the processor leaves and returns to an idle state, we can measure the time it takes to handle an interactive event, and the time during which a user might be waiting.

The lack of kernel source code prevents us from instrumenting the kernel to identify the exact times at which

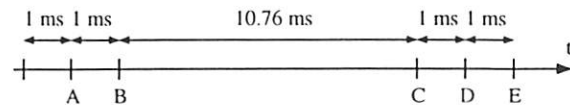


Figure 1. Validation of Idle-Loop Methodology. The system spent one ms collecting each of samples A, B, D, and E but spent 10.76 ms collecting sample C, indicating the system performed 9.76 ms of work during this interval.

the processor leaves or enters the idle loop. Instead, we replace the system's idle loop with our own low-priority process in each of the operating systems.² These low-priority processes measure the time to complete a fixed computation: N iterations of a busy-wait loop. The instrumentation code logs the time required by the loop. The pseudo code is as follows:

```
while (space_left_in_the_buffer) {
    for (i = 0; i < N; i++)
        ;
    generate_trace_record;
}
```

We select the value of N such that the inner loop takes one ms to complete when the processor is idle. In this way we generate one trace record *per millisecond of idle time*. If the processor is taken away from the idle loop, the loop takes longer than one ms of elapsed time to complete. Any non-idle time manifests itself as an elongated time interval between two trace records. The larger we make N , the coarser the accuracy of our measurements; the smaller we make N , the finer the resolution of our measurements but the larger the trace buffer required for a given benchmark run.

We wrote and measured a simple microbenchmark to demonstrate and validate this methodology. It uses a program that waits for input from the user and when the input is received, performs some computation, echoes the character to the screen, and then waits for the next input. We measured the time it took to process a key-stroke in two ways. First, we used the idle loop method described above to measure the processing time. Figure 1 shows the times at which the samples were collected. For the sake of clarity only a few samples are

2. Since both versions of the Windows NT kernel were preemptible, we were able to run the measurement loop inside the kernel. We executed the measurement loop from user-level under Windows 95 since its kernel is not preemptible. This requires our measurement code to make a system call to access the performance counters every time the measurement code generates a sample. All of our results have been corrected for this extra overhead.

shown. The figure shows that the system spent approximately one ms generating samples A, B, D, and E, indicating that the system was idle during the periods in which these samples were generated, but spent 10.76 ms generating sample C. The difference, $(10.76 - 1)$ or 9.76 ms, represents the time required to handle the event.

Next, we used the traditional approach, recording one timestamp when the program received the character (i.e., after a call to `getchar()`) and a second timestamp after the character was echoed back to the screen. This measurement reported an event-handling latency of only 7.42 ms. The 2.34 ms discrepancy between the two measurements highlights a shortcoming of the conventional measurement methodology. Our test program calls the `getchar()` function to wait for user input. When the user enters a character, the system generates a hardware interrupt, which is first handled by the dynamically linked library `KERNEL32.DLL`. In the traditional approach, the measurement does not start until control is returned to the test program. Therefore, it fails to capture the system time required to process the interrupt and reschedule the benchmark thread. In comparison, our idle loop methodology provides a more complete measurement of the computation required to process the keyboard event.

Our idle loop methodology uses CPU busy time to represent event latency, but there are several issues that prevent this from being an accurate measure of the user's perceived response time. One problem is that most graphics output devices refresh every 12–17 ms. In this research, we do not consider this effect.

Another problem is that CPU busy time and CPU idle time do not equate directly with wait time and think time. First, synchronous I/O requests contribute to wait time, even though the CPU can be idle during these operations. Second, in the case of background processing, the user may not be waiting even though the CPU is busy. The first problem could be solved with system support for monitoring the I/O queue and distinguishing between synchronous and asynchronous requests. In order to address the second problem, we must consider how events are processed by the systems. When the user generates key strokes and mouse clicks, they are queued in a message queue to await processing. Therefore, when there are events queued, we can assume that the user is waiting. By combining CPU status (busy or idle), message queue status (empty or non-empty), and status for outstanding synchronous I/O (busy or idle), we can speculate during which time intervals the user is waiting.

Figure 2 shows a state transition diagram for identifying think time and wait time in our system, using the parameters: CPU state, message queue state, and synchronous I/O status. The diagram omits asynchronous I/O, which we assume is background activity, and assumes that users always wait for the completion of an event. In reality, we can never precisely distinguish think time from wait time, because we cannot know what the user is doing and whether the user is actually waiting for an event to complete or is thinking while an event is being processed. For simplicity, in the rest of this paper, we assume that the user waits for each event and report results in terms of event handling latency. In the next

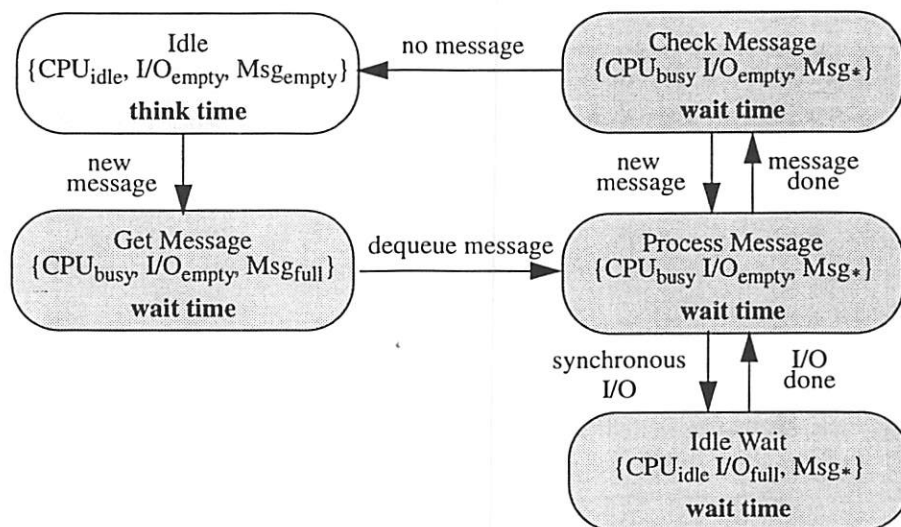


Figure 2. System State Transition Diagram. Monitoring CPU busy time, the I/O queues, and the message queue provides a framework for identifying wait time and think time. We assume that asynchronous I/O is always background activity (and is not shown) and that synchronous I/O causes long-latency events for which users wait.

section, we describe how we obtain information about the status of the message queue.

2.4 Monitoring the Message API

Win32 applications use the `PeekMessage()` and `GetMessage()` calls to examine and retrieve events from the message queue. We can monitor use of these API entries by intercepting the `USER32.DLL` calls. By monitoring use of these API entries, we can detect when an application is prepared to accept a new event and when it actually receives an event. We correlate the trace of `GetMessage()` and `PeekMessage()` calls with our CPU profile to determine when the application begins handling a new request and when it completes a request. This allows us to distinguish between synchronous and asynchronous I/O. It is also useful for recognizing situations where asynchronous computation is used to improve interactive response time.

Figure 2 illustrates our design for a finite state machine that distinguishes think time from wait time in a latency-measurement system. In Sections 4, 5, and 6, we will demonstrate how to apply complete information about CPU state and partial information about message queue state to implement part of the FSM. Implementation of the full FSM requires additional system support for monitoring I/O and message queue state transitions. Implementation of such monitoring is part of our continuing work at Harvard.

Next, we will present two simple example measurements to give some insight into some of the non-trivial aspects of interpreting the output of our measurements.

2.5 Idle System Profiles

In this section, we present measurement results for the background activity that occurs during periods of inactivity on Windows NT and Windows 95. This provides intuition about the measurement techniques as well as baseline information, useful for interpreting latency measurements in realistic situations. Figure 3 shows the idle system profiles for the three test systems. To relate non-idle time to elapsed time, we plot elapsed time on the X-axis and the CPU utilization on the Y-axis. Given that each sample represents 1 ms of idle time, the average CPU utilization during a sample interval can be calculated easily. For example, if the system spends 10 ms collecting a sample, and the sample includes 1 ms of idle time, the CPU utilization for that time interval is $(10 - 1)/10 = 90\%$.

Both versions of Windows NT show bursts of CPU activity at 10 ms intervals due to hardware clock interrupts. Correlating the samples with a count of hardware

interrupts from the Pentium performance counters shows that each burst of computation is accompanied by a hardware interrupt.

Although we have compensated for the overhead introduced by the user-level idle loop, Windows 95 shows a higher level of activity in comparison to both versions of the NT system. We do not know what causes this increased activity in Windows 95.

By coupling our idle-loop methodology with the Pentium counters, we were able to compute the interrupt handling overhead for various classes of interrupts — measurements difficult to obtain using conventional methods. For example, the smallest clock interrupt handling overhead under Windows NT 4.0 was about 400 cycles, or 4 μ s.

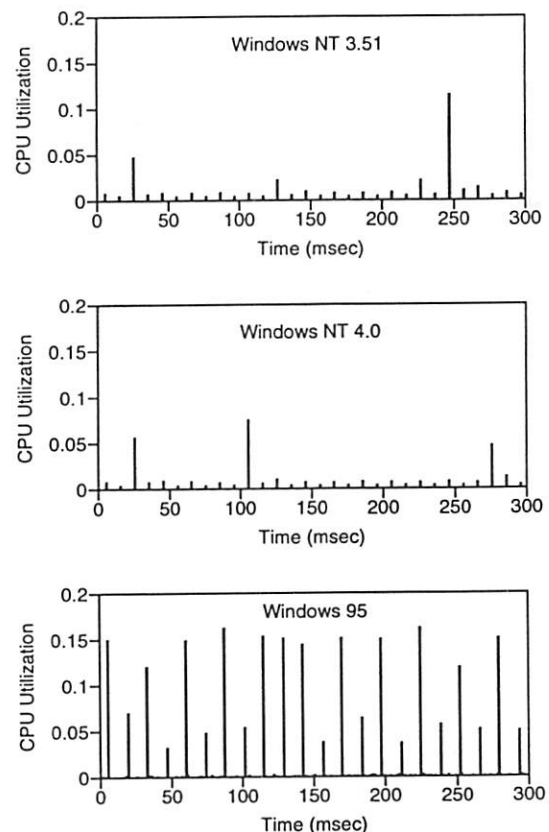


Figure 3. Idle Loop Measurements for the three operating systems. These graphs provide a graphical representation of our idle-loop measurement and illustrate differences in the idle behavior of the three systems. For each recorded sample, we subtract the one ms of idle time that our busy-wait loop requires to obtain the amount of non-idle activity. During an idle period, this difference is zero, and we report a CPU utilization of zero. When there is other CPU activity, more than one ms of time will pass before the completion of one ms of idle time, and the CPU utilization during that interval is non-zero.

2.6 Window Maximize

It is becoming common for interactive systems to perform extra work, such as animation, for the user's "viewing pleasure." The following experiment shows the effects of such features. We measured the system activity generated when maximizing a previously minimized window; the system displays a window that increases in size gradually as it is maximized. In order for the animation to be visible to the user, the system must pace itself, inserting delays. These delays cause the system to become idle, resulting in multiple idle-loop measurement samples that correspond to the single user event (i.e., maximize window).

Figures 4a and 4b show the results of our measurement in two different resolutions. Figure 4a shows the full 1 ms resolution of the data, while Figure 4b shows the CPU utilization averaged over 10 ms intervals. Both figures clearly show the 80 ms of 100% CPU utilization required to process the input event (from 100 to 180 ms) and another period of 100% CPU utilization starting around 400 ms to redisplay the page. The stair pattern between 180 and 400 ms illustrates the CPU activity required to perform the animation. From Figure 4a, we

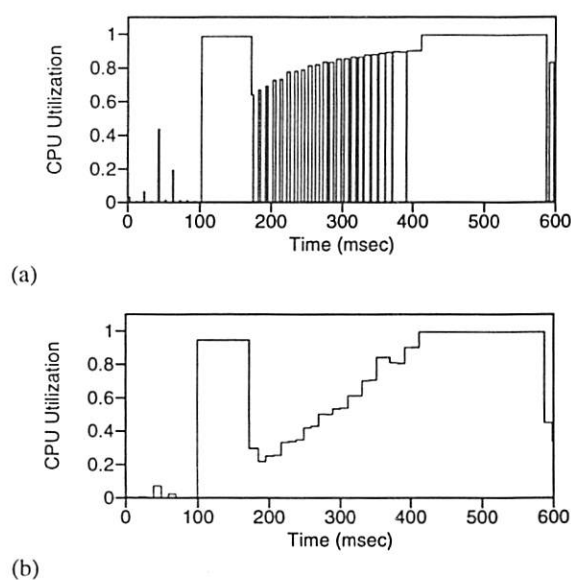


Figure 4. CPU Usage Profile for a Window Maximize Operation under Windows NT 4.0. Figure 4a shows the measurement of computation generated when a window is transformed from an icon in the NT 4.0 icon bar to an open window using 1 ms resolution. The short spikes between 180 ms and 400 ms correspond to the animation that is used as the window outline moves from the menu bar to its destination. The upward slope occurs as the outline increases in size. The period of continuous computation from 400 ms to 600 ms corresponds to the redraw computation for the window.

Figure 4b shows the same data as Figure 4a, but with the CPU utilization averaged over 10 ms intervals. The details in Figure 4a are useful, but become illegible for longer running tests.

can observe that the bursts of CPU activity for performing animation are aligned on 10 ms boundaries, suggesting that they are scheduled by clock interrupts. Each step of animation takes progressively longer time to complete as the window outline increases in size.

This measurement shows that a single user event can correspond to multiple intervals of CPU busy time. Such events complicate the task of precisely identifying event boundaries. Monitoring the Message API (section 2.4) is one of the techniques that helps us pinpoint the beginning and ending of interactive events.

3 Benchmarks and Metrics

Our benchmark set is organized into three categories. *Microbenchmarks* are useful for understanding system behavior for simple interactive operations, such as interrupt handling and user-interface animation. By analyzing microbenchmarks, we develop an understanding of the low-level behavior of the system. We then extend our measurement to *task-oriented benchmarks* in order to understand the real impact of latency on the perceived interactive responsiveness of an application. These task-oriented benchmarks are based on applications from typical PC office suites and are designed to represent a realistic interactive computing workload. We further apply *application microbenchmarks* to evaluate isolated interactive events from the realistic workloads. Our application microbenchmarks include such computations as page-down of a Power-Point document and editing of an embedded OLE object.

We used Microsoft Visual Test to create most of our microbenchmarks and task-oriented benchmarks. MS Test provides a system for simulating user input events on a Windows system in a repeatable manner. Test scripts can specify the pauses between input events, generating minimal runtime overhead. However, in some cases, the way that Test drives applications alters the behavior of those applications. This effect is discussed in detail in Section 5.4.

3.1 Evaluating Response Time

Early in this project, we had planned to develop a new latency metric, a formula that could be used to summarize our measurements and provide a single scalar figure of merit to characterize the interactive performance of a given workload. Events that complete in 0.1 seconds or less are believed to have imperceptible latency and do not contribute to user dissatisfaction³, whereas events in the 2-4 second range invariably irritate users [12]. Events that fall between these ranges may be acceptable but can correspond to perceptibly worse performance than events under 0.1 seconds. Our intuition is that a

user-responsiveness metric would be a summation of the form:

$$\sum_i f_i(x) \quad \text{where}$$

$$f_i(x) = \begin{cases} 0 & t(i) \leq T \\ (t(i) - T)^p & t(i) > T \end{cases}$$

T = user perception threshold
 p = some exponent
 $t(i)$ = latency of event i

However, we also believe that the threshold, T , is a function of the type of event. For example, users probably expect keystroke event latency to be imperceptible while they may expect that a print command will impose some delay. The issues of event types, user expectation, the precise tolerance of users for delay, and the limitations of human perception are beyond our field of expertise. Presented with these obstacles, we modified our plans, and present latency measurements graphically. We trust that the issues in human-computer interaction can be resolved by specialists. In the meantime, our visualization of latency enables us to compare applications and develop an intuition for responsiveness without risking the inappropriate data reductions that could occur given our limited background in experimental psychology.

3.2 Visualizing Event Latency

Figure 5 is an example of the graphical representation of our raw data. Each vertical bar represents an event that began at the time represented by the X value and lasted for a period represented on the Y axis. Figure 5a shows the data for an entire Microsoft Word benchmark run, while 5b shows a magnification of a two second interval. The complete event latency profile provides a very coarse view of the application, while the magnification provides the detail to explain the periodicity in the overall pattern. By drawing a horizontal line at a given "irritation threshold," the frequency and distribution of irritating events is readily visible.

For our task-oriented benchmarks, we use three graphical representations to capture the responsiveness of an application. First, we present histograms, showing the number of events corresponding to each measured latency. This presents a detailed breakdown of the event latencies and provides some intuition into the different categories of events present in an application. Next, we

3. For multimedia applications and applications that use visual feedback (such as rubber-banding in a drawing program), latency requirements may be smaller. The limitations of the human visual system suggest that 10 ms provides a lower bound on the latencies that can be perceived for continuous media operations [6].

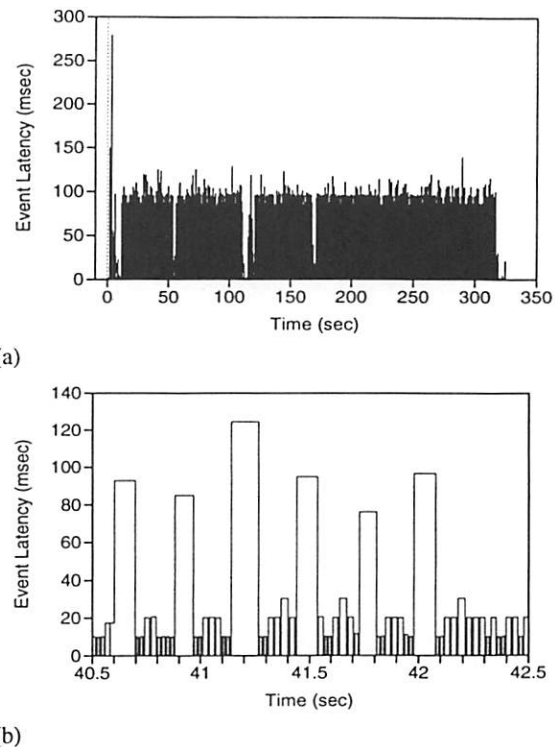


Figure 5. Raw Data Representation. Figure 5a represents a 1000 event trace of Microsoft Word on Windows NT 3.51. The benchmark is sufficiently long that a complete profile does not lend itself to detailed analysis. The magnified profile (b) shows the periodicity of long and short latency events. Notice that the majority of the events fall below the 0.1 second threshold of user perception but that a significant number fall well above the threshold.

integrate over the histogram presenting a cumulative latency graph. This provides the quantitative data indicating how events of a particular duration contribute to the overall time required to complete a task. Finally, we plot the cumulative latency as a function of the number of events, providing an intuition about the variance in response time perceived by the user. Note that in each of these cases, the events are sorted by their duration, not by their actual time of occurrence.

4 Microbenchmarks

In this section, we present some basic measurements of simple interactive events. This helps us explore the character of our tools and understand the kinds of things we can and cannot measure.

Figure 6 shows the latencies for two simple interactive events, unbound key stroke and mouse click on the screen background, under the three operating systems. We were unable to measure the overhead of Microsoft Test for these microbenchmarks, so we were forced to use manual input. To compensate for the potential variability introduced by a human user, we report the mean

of 30–40 trials, ignoring cold cache cases. The most significant standard deviations occurred in the key click events for Windows NT 4.0 and Windows 95 (8%) while all the remaining standard deviations were under 2% of the mean.

On the key stroke test, Windows 95 shows substantially worse performance than NT 4.0. This is a reflection of segment register loads (not shown) and other overhead associated with 16-bit windows code [4], which persist in Windows 95.

The mouse click results are even more striking. The Windows 95 measurements are off the scale, because the system busy-waits between “mouse down” and “mouse up” events; therefore our measurement indicates the length of time the user took to perform the mouse click. This is much longer than the actual processing times of the NT systems and is not indicative of the actual Windows 95 performance.

Our methodology provides little guidance in explaining the differences in performance between Windows NT 3.51 and NT 4.0, but it does highlight the fact that instructions and data references occur roughly in proportion to cycles across the systems for both of the simple interactive events. Therefore, we conclude that in the warm cache case, the performance differences are a function of the code path lengths. It is possible that the difference in code path length stems from the change in GUI between NT 3.51 and NT 4.0.

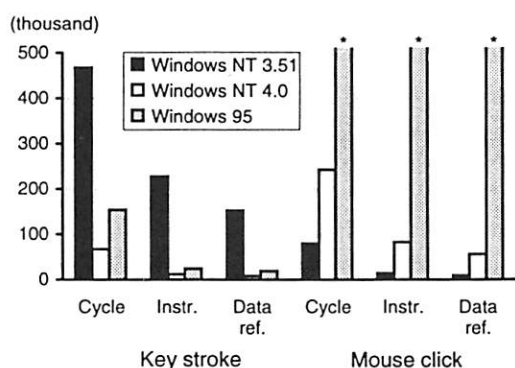


Figure 6. Latency for simple interactive events. This figure shows the latencies for a simple mouse click and a simple key stroke in each of the three systems. In Windows 95, the system remains busy the entire time between a “mouse down” and a “mouse up” event; therefore the latency of the event is a function of the duration of the user’s press on the mouse. This is much larger than the actual processing times of the NT systems and is not indicative of Windows 95 performance. Each microbenchmark was repeated 30–40 times; the standard deviations were less than 8% for key stroke on Windows NT 4.0 and Windows 95 and less than 2% on the remaining tests.

5 Task-oriented Benchmarks

In this section, we measure three task-oriented benchmarks, designed to model realistic tasks that users commonly perform using the target applications. In using these longer running benchmarks we have two specific goals. The first is to measure the system performance for a realistic system state. An often-cited problem of microbenchmarks is that they tend to measure the system when various caches are already warm. However, measuring the system when all the caches are cold is also unrealistic. Neither extreme is representative of the system state in which the target micro-operations are invoked in common practice. By measuring the latency of micro-operations embedded in a longer realistic interactive task, we measure each micro-operation under more realistic circumstances. The second goal is to identify long-latency operations that users encounter as they perform tasks on the systems. Since these long-latency operations have a greater effect on how users perceive system performance than very short events, we will isolate several such operations in our latency record and examine their activity in detail.

We ran each benchmark five times using Microsoft Test and found that the results were consistent across runs. The standard deviations for the elapsed times and cumulative CPU busy times were 1–2%, and the event latency distributions were virtually identical. The graphical output shown in the following sections depicts one of the five runs for each benchmark.

5.1 Microsoft Notepad

Notepad is a simple editor for ASCII text distributed with all versions of Microsoft Windows. Our Notepad benchmark models an editing session on a 56KB text file, which includes text entry of 1300 characters at approximately 100 words per minute, as well as cursor and page movement. With this benchmark, we demonstrate how differences in average response time across the three systems manifest themselves in our visual representation of latency and how they can be used to compare system performance. We used the same Notepad executable (the Windows 95 version) on all three systems and used a Microsoft Test script to drive Notepad. Since virtually all Notepad activity is synchronous, we were able to collect the latency figures for every keystroke that the user made in a straightforward way. By correlating our idle loop measurement with our monitoring of the PeekMessage() and GetMessage() API calls, we were able to clearly identify the Test overhead and remove it from the data presented in Figure 7.

Notice that the Y scale in the histogram in Figure 7 is a logarithmic scale. The cumulative latency graph shows

that for all three systems, over 80% of the latency of Notepad is due to low-latency (less than 10 ms) events. These short-latency events are the keystrokes that generate printable ASCII characters. The remaining 20% of the total latency are due to the longer latency (at least 28 ms) keystrokes that cause "page down" or newline operations. These keystrokes cause Notepad to refresh all or

part of the screen. The smoothness of the curves in the bottommost graph in Figure 7 shows that there is little variance in either the long latency events or the short-latency events. Events of the same type contribute equally to the total latency.

The latencies measured are relatively small for Notepad and reflect both the simplicity of the application and the relatively fast PC that we used for our experiments. Although these differences in latency are likely to go unnoticed by users of our test system, they might have a significant effect on user-perceived performance on a slower machine.

5.2 Microsoft Powerpoint

Powerpoint, from the Microsoft Office suite, is a popular application for creating presentation graphics. In our Powerpoint task scenario, the user starts Powerpoint immediately after powering up the machine and booting the operating system, so that all caches are cold. The user then loads a 46-page, 530KB presentation, and finds and modifies three OLE embedded Excel graph objects⁴. Each of the OLE objects was of similar size and complexity. As with Notepad, we used a Microsoft Test script to drive the application and deliver keystrokes at a realistic rate, with each keystroke separated by at least 150 ms. An important property of the Powerpoint benchmark is that it has a number of events with easily perceptible latencies. Since we were mainly interested in longer events, we pre-processed our data to exclude events with latency of less than 50 ms. Figure 8 shows the results for the two versions of Windows NT. We were unable to run this experiment for Windows 95 due to limitations of Microsoft Test when manipulating OLE embedded object on that system.

The shortest events in Figure 8 (with latency of less than one second) are due to "page down" operations and Excel operations. Both systems exhibited a similar latency distribution for these events. Six events had latencies greater than one second on both systems, in nearly the same relative order. Table 1 lists these long latency events.

All of the long-latency events required disk accesses, which are responsible for the majority of the latency for these events. The effects of the file system cache are most clearly observed in the latency for starting the second OLE edit, as more of the pages for the embedded Excel object editor become resident in the buffer cache.

4. Each OLE edit requires starting the server application (in this case Excel), editing the object, and exiting the server application.

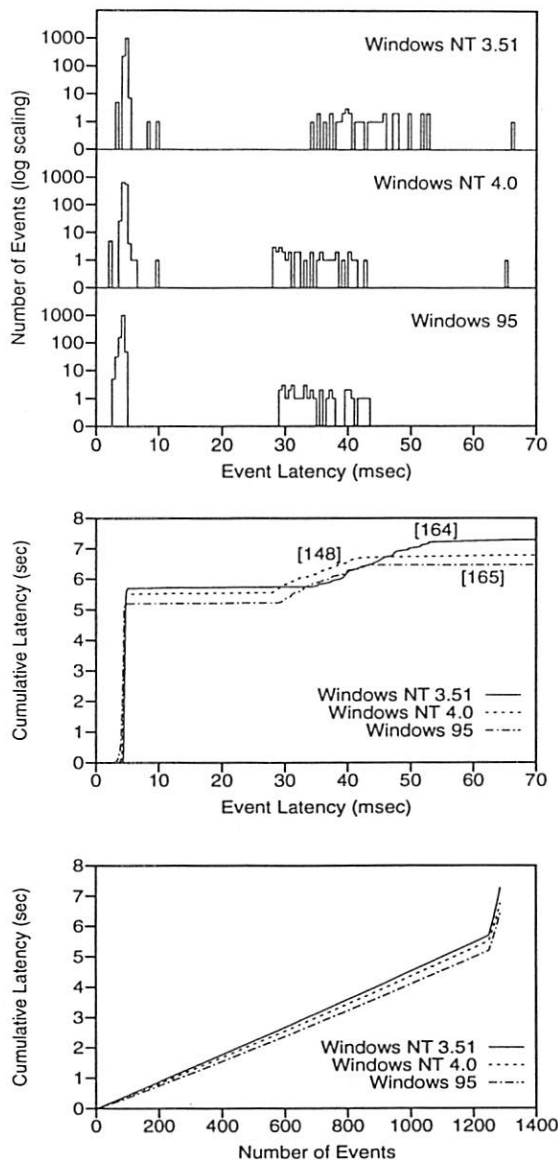


Figure 7. Notepad Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. Notice that Windows 95 has the smallest cumulative latency, but its elapsed time is larger than that of both NT 3.51 and NT 4.0. This seemingly anomalous result is an artifact of Microsoft Test, which generates an extra WM_QUEUE_SYNC message after each input event. The time to process this message is longer under Windows 95 than under the NT systems. We do not include this time in the event latencies, but it does contribute to elapsed time.

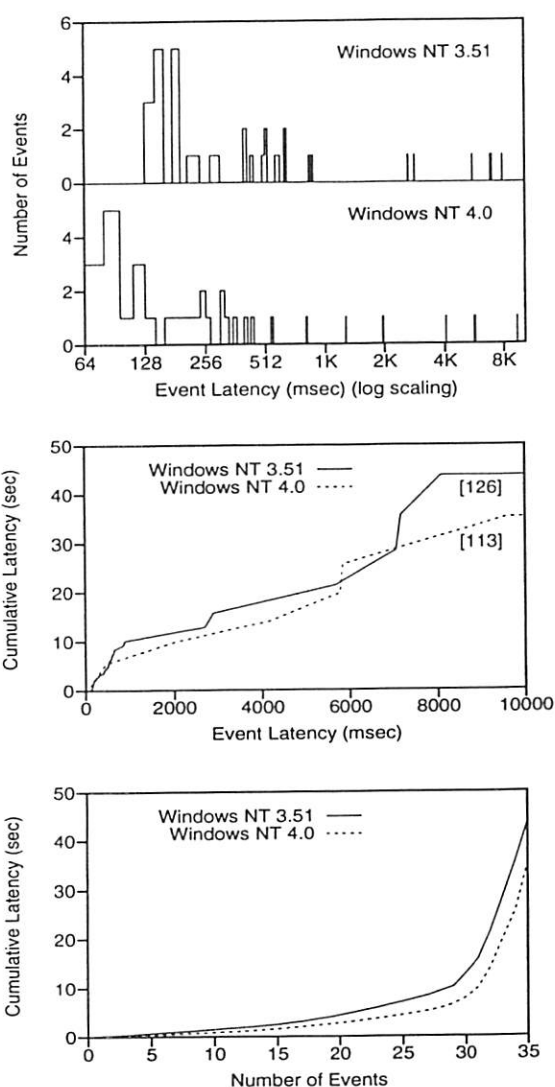


Figure 8. Powerpoint Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. While most of the events for Powerpoint are relatively short (under 500 ms), the majority of the time is spent in long-latency events.

The cumulative latency graph shows that both versions of Windows NT demonstrate similar performance for the short-latency keystrokes, and the majority of the performance difference is a result of the ability of NT 4.0 to handle the long-latency events much more efficiently. We turn to application microbenchmarks to examine this phenomenon in more detail.

5.3 Powerpoint Microbenchmarks

Our Powerpoint task reveals significant differences in latency between the two version of NT but Powerpoint's complexity makes it difficult to reveal the source of these differences. We designed two application

	latency (in seconds)	
	NT 3.51	NT 4.0
Save document	8.082	9.580
Start Powerpoint	7.166	5.773
Start OLE edit session (first time)	7.050	5.844
Open document	5.680	4.151
Start OLE edit session (second object)	2.897	2.009
Start OLE edit session (third object)	2.697	1.305

Table 1. Powerpoint events with latency over one second.

microbenchmarks to explore page down and OLE edit operations in detail. Both operations have a perceptible response time and occur multiple times during our Powerpoint task. We measured each operation using Test and collected data from the Pentium performance counters to look for hints to identify the performance differences between the systems. Because of the simplicity of the application microbenchmarks, we were able to run them on the Windows 95 systems as well as the two NT systems.

Page Down

For this benchmark, we measured the latency and the hardware events that occur during a "page down" to a page containing an OLE embedded graph. We repeated the test 10 times for each performance counter, so our measurements reflect warm cache behavior. The standard deviations are all below 3%. Figure 9 shows some of the hardware events observed in each system during the page-down operation. The graph shows that NT 4.0 was able to handle the request in the shortest amount of time followed by Windows 95 and NT 3.51. The difference in the latency between the two versions of Windows NT is explained by the differences in system architecture. In NT 3.51, the Win32 API is implemented by a user-level server. The negative performance effects of this server-based architecture were demonstrated in prior research [4]. In NT 4.0, some components of the Win32 API server are rumored to have been moved into the kernel. The improved locality from this change is reflected in reduced TLB misses for NT 4.0 compared to NT 3.51. A lower TLB miss rate implies fewer protection domain crossings in Pentium processors, which flush the TLB on each crossing [5]. Using 20 cycles per miss as a lower bound on TLB miss handling latency⁵, the extra TLB misses that occur for NT 3.51 (both

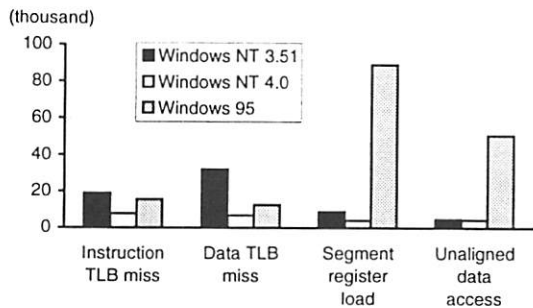
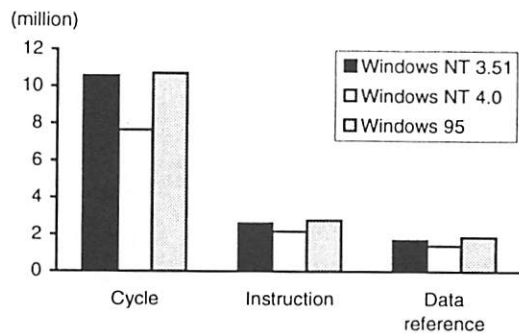


Figure 9. Counter Measurements for Powerpoint page down operation. The larger number of TLB misses in NT 3.51 relative to NT 4.0 indicates a greater number of protection domain crossings. The large number of segment register loads for Windows 95 results from code that runs in 16-bit mode and accounts for the majority of the difference between Windows 95 and the NT systems.

instruction and data) account for at least 25% of the latency difference between NT 3.51 and NT 4.0.

Comparing NT 4.0 and Windows 95, there are a relatively large number of unaligned data accesses and segment register loads for Windows 95 (Figure 9). The high counts for these events are due to the large components of Windows 95 (such as the graphics API) that are implemented in 16-bit code. Windows 95 also incurs 93% more TLB misses than NT 4.0, although we do not have sufficient information to attribute this behavior to a specific architectural feature in the two systems.

OLE Edit

The long latency events in our Powerpoint task suggest that much of the OLE edit start-up latency is due to disk

5. Our idle loop measurements give us a good lower bound on the TLB miss penalty. By monitoring both the TLB miss count and elapsed cycles, we can compare measurement intervals where one TLB miss occurred to intervals where no TLB misses occurred. Twenty cycles is the lowest TLB miss handling time observed during our measurements, and would require that both of the memory references required to handle a TLB miss on the Pentium hit in the first level cache.

I/O. In this section, we exclude the effects of disk I/O by measuring an OLE edit start-up with a hot buffer cache. We also repeated this test 10 times for each counter, but we noticed that all of the events and the cycle counter increased steadily on subsequent runs. We speculate that this behavior is unintended, and therefore, report the results from the first run. Figure 10 shows some of the hardware event counts during the OLE edit start-up. Like the page down benchmark, NT 4.0 completes the operation with the shortest latency followed by Windows 95 and NT 3.51. Comparing NT 4.0 to NT 3.51, the observations made for the page down benchmark carry through to the OLE edit. Overhead from elevated TLB miss rates account for at least 23% of the latency difference between NT 3.51 and NT 4.0. In Windows 95, we observed a large number of segment register loads and unaligned data accesses, both of which can be attributed to code executing in 16-bit mode.

5.4 Microsoft Word

Our task-oriented workload for Microsoft Word consists of text entry of a paragraph of approximately 1000 characters. It includes cursor movement with arrow keys and backspace characters to correct typing errors. The timing between keystrokes was varied to simulate realistic pauses when composing a document, and line justification and interactive spell checking were enabled. We do

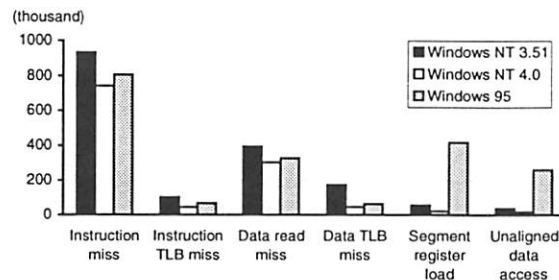
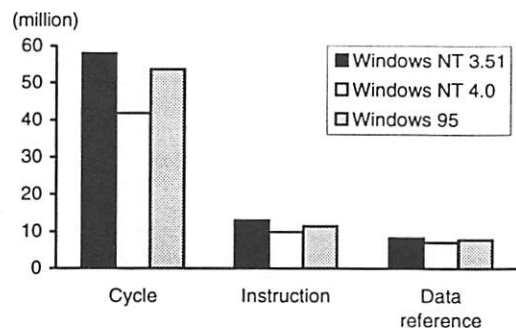


Figure 10. Counter Measurements for Powerpoint OLE edit start-up operation. As in Figure 9, TLB misses account for a significant fraction of the difference between NT 3.51 and NT 4.0, and segment register loads account for the majority of the difference between Windows 95 and NT.

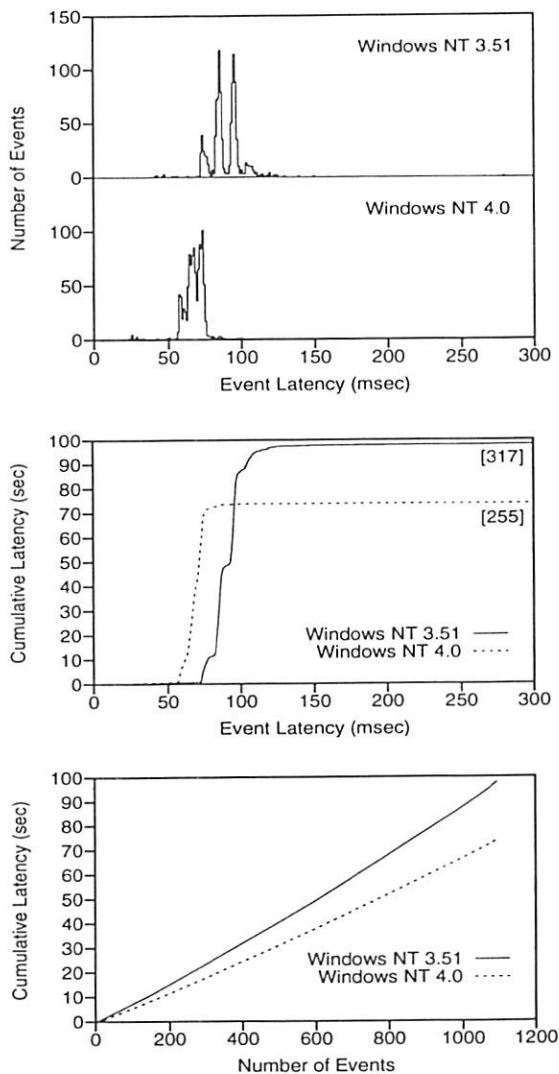


Figure 11. Microsoft Word Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. Although NT 4.0 demonstrates uniformly better response time, both systems have most latencies below the threshold of user perception.

not report results for Windows 95, because the system does not become idle immediately after Word finishes handling an event, making all event latencies appear to be several seconds long.

Figure 11 shows results for Microsoft Test driven simulations on the two versions of Windows NT. Compared to Notepad, Word requires substantially more processing time per keystroke, due to additional functionality such as text formatting, variable-width fonts and interactive spell checking. For the majority of interactive events, NT 4.0 exhibits shorter response time and lower variance than NT 3.51.

The Microsoft Word benchmark demonstrates both the

strengths and limitations of evaluating interactive performance using latency. Compared to throughput measurements, our latency analysis provides much more detailed information, such as variations in latency and the distribution of events with different latencies. However, the structural features of Word push us to the limit of the behavior we are able to analyze. Our analysis indicates that Word uses a single system thread, but responds to input events and handles background computations asynchronously using an internal system of coroutines or user level threads.

Distinguishing background activity from foreground activity in Word is challenging. We examined the results of hand-generated Word input under Windows NT 3.51, compared it to the Test-generated results, and found significant differences. For our hand-generated tests, we ran seven trials, with the same typist and input, and found that the event histograms appeared very similar and that the variation in cumulative latency and elapsed time was less than 4% across the runs. While the Test results showed that most events had latency between 80 and 100 ms, we measured a 32 ms typical latency for the hand-generated input. This difference in event latency was accompanied by a compensating difference in background activity. The hand-generated input showed a higher level of background activity than the Test-generated results. We also observed that carriage returns under the hand-generated input took longer than 200 ms to handle while the longest latency events we saw in the Test-generated runs were 140 ms. Our Message API log reveals that Test generates a WM_QUEUESYNC messages after every keystroke. We hypothesize that these messages were responsible for the different behavior under Test and under manual typing. However, with our current tools, the complexity of Word makes it difficult to thoroughly analyze even the simple experiment we present here.

6 Discussion

The tools and techniques we have discussed here are a first step towards understanding and quantifying interactive latency, but there remains much work to be done. In the absence of system and application source, better performance monitoring tools would be useful. Our measurements could be improved through API calls that return information about system state such as message queue lengths, I/O queue length, and the types of requests on the I/O queue. Currently, some of this information can be obtained, but it is painful (e.g., monitoring the GetMessage() and PeekMessage() calls).

Even in the presence of rich APIs, the task of distinguishing between wait time and think time is not always

possible. There is no automatic way to detect exactly what a user is doing. Without user input, we can never tell whether a user is genuinely waiting while the system paints a complicated graphic on the screen or is busy thinking. For simulations using designed scripts, we can make assumptions about when users think and then analyze performance based on those assumptions, but the most useful analysis will come from evaluating actual user interaction.

One factor that contributes to user dissatisfaction is the frequency of long-latency events. We processed the Microsoft Word profile of Figure 5 to analyze the distribution of interarrival times of events above a given threshold. Since most events in the Word benchmark were very short, we chose thresholds around 100 ms. Table 2 shows the summaries for these thresholds. Note that the standard deviations are of the same order of magnitude as the averages themselves, indicating that there is no strong periodicity between long-latency events.

Threshold (in msec)	Number of events above threshold	Interarrival times	
		Average (in sec)	Std. Dev. (in sec)
100	101	3.1	3.1
110	26	12.4	10.6
120	8	41.1	48.8

Table 2. Summary of interarrival distributions for Microsoft Word benchmark on Windows NT 3.51. Notice that an increase of 10% in the threshold (from 100 ms to 110 ms) reduces the number of above threshold events by a factor of 4.

We then examined the truly long-latency events from the Powerpoint benchmark. Figure 12 shows the event latency profile for all events over 50 ms. Both systems show similar periodicity with the better performing 4.0 system demonstrating smaller interarrival times to match its shorter overall latency.

In the case of Word, the interarrival times are clustered because most events have similar latency. In the case of Powerpoint, the interarrival times of long-latency events are simply the interarrival times of a few particular classes of events. The distribution of these events is entirely dependent upon when we issued such requests in our test script and is not necessarily indicative of the distribution that might be obtained from a real user. In this test, none of the simple keystroke events were responsible for generating long-latency events, rather all the events with latencies over 50 ms result from major operations for which user expectation for response time

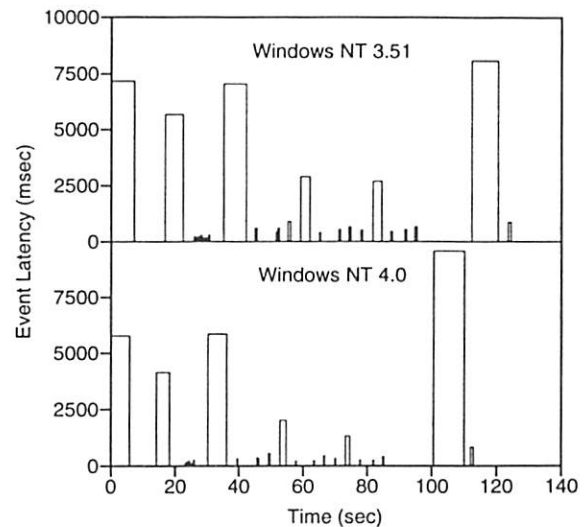


Figure 12. Time Series of long-latency events for Powerpoint. Both systems show similar distributions, with NT 4.0 having slightly shorter interarrival intervals.

is generally longer. Until our tools become sophisticated enough to examine long traces of complex events generated by a real user, further analysis of these interarrival times is not particularly productive.

Over time, our tools will become better able to deal with the sophisticated applications that we seek to analyze, but we need the human factors community to assist us in understanding the limits of human perception and the models of user tolerance. Some of the questions that must be answered are:

- What are the limits of human perception?
- How do the limits vary by task (e.g., typing versus mouse-tracking)?
- How do the user expectation and tolerance for interactive response time vary by task?
- How does user dissatisfaction grow with increasing of latency?
- How does user dissatisfaction grow with the variance of latency?
- What aspects of performance contribute the most to user satisfaction?

7 Conclusions

Latency, not throughput, is the key performance metric for interactive software systems. In this paper, we have introduced some tools and techniques for quantifying latency for a general class of realistic interactive application. To demonstrate our methodology, we applied it to compare the responsiveness of realistic applications running on three popular PC operating systems.

Whereas current measurements of latency are generally limited to microbenchmarks, our approach allows us to measure latency for isolated events in the context of realistic interactive tasks. Our latency measurements give a more accurate and complete picture of interactive performance than throughput measurements.

We have combined a few simple ideas to get precise information about latency in interactive programs. We have shown that using these ideas we can get accurate and meaningful information for simple applications and also, to a degree, for complex applications. The requirements of these techniques are not out of reach; in particular, a hardware cycle counter, a means for changing the system idle loop, and a mechanism for logging calls to system API routines are needed. Additional support for detecting the enqueueing of messages and the state of the I/O queue would provide a more complete framework for latency measurement. We have shown the limitations of our system for applications such as Microsoft Word that use batching and asynchronous computation.

Measuring latency for an arbitrary task and an arbitrary application remains a difficult problem. Our experience with Microsoft Word demonstrates that there are many difficult technical issues to be resolved before latency will become a practical metric for system design. Our graphical representation provides a great deal of information about program behavior to specialists, but is probably not appropriate for more widespread use. The two key components necessary to provide consumers a single figure of merit are further work in human factors and some method for distinguishing user think time from user wait time.

8 References

- [1] Business Applications Performance Corporation, "SYSmark for Windows NT," Press Release by IDEAS International, Santa Clara, CA, March 1995.
- [2] Brian N. Bershad, Richard P. Draves, and Alessandro Forin, "Using Microbenchmarks to Evaluate System Performance." *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, Key Biscayne, Florida, April 1992, pages 148–153.
- [3] Ben Smith, "Ultrafast Ultrasparcs," *Byte Magazine*, January 1996, page 139. Additional information on the Bytemarks suite is available on the Internet: <http://www.byte.com/bmark/bdoc.htm>.
- [4] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael D. Smith, "The Measured Performance of Personal Computer Operating Systems," *ACM Transactions on Computer Systems* 14, 1, February 1996, pages 3–40.
- [5] Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual*, Intel Corporation, 1995.
- [6] C. J. Lindblad and D. L. Tennenhouse, "The VuSystem: A Programming System for Compute-Intensive Multimedia," To appear in *IEEE Journal of Selected Areas in Communication*, 1996.
- [7] Larry McVoy, "Lmbench: Portable tools for performance analysis," *Proceedings of the 1996 USENIX Technical Conference*, January 1996, pages 179–294.
- [8] Jeffrey C. Mogul, "SPECmarks are leading us astray," *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, Key Biscayne, Florida, April 1992, pages 160–161.
- [9] James O'Toole, Scott Nettles, and David Gifford, "Concurrent Compacting Garbage Collection," *The Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993, pages 161–174.
- [10] John K. Ousterhout, "Why Operating Systems Aren't Getting Faster As Fast As Hardware." *Proceedings of the Summer 1991 USENIX Conference*, June 1991, pages 247–256.
- [11] Mark Shand, "Measuring Unix Kernel Performance with Reprogrammable Hardware," Digital Paris Research Lab, Research Report #19, August 1992.
- [12] Ben Shneiderman, *Designing the User Interface*, Addison-Wesley, 1992.
- [13] Jeff Reilly, "SPEC Discusses the History and Reasoning behind SPEC 95," *SPEC Newsletter*, 7(3):1–3, September 1995.
- [14] M. L. VanName and B. Catchings, "Reaching New Heights in Benchmark Testing," *PC Magazine*, 13 December 1994, pages 327–332. Further information on Ziff-David benchmarks is available on the Internet: <http://www.zdnet.com/zdbop/>.

Dynamic Binding for an Extensible System

Przemysław Pardyak and Brian N. Bershad

{pardy,bershad}@cs.washington.edu

Department of Computer Science and Engineering

University of Washington

Seattle, WA 98195, USA

Abstract

An extensible system requires a means to dynamically bind extensions into executing code. The *SPIN* extensible operating system uses an event-based invocation mechanism to provide this functionality in a flexible, transparent, safe, and efficient way. Events offer a uniform model of extensibility, whereby the system's configuration can change without changing any of its components. Events are defined with the granularity and syntax of procedures but provide extended procedure call semantics such as conditional execution, multicast, and asynchrony. By installing a handler on an event, an extension's code can execute in response to activities at the granularity of procedure call. Our system uses runtime code generation to ensure that event delivery has low overhead and scales well with the number of handlers. This paper describes the design, use and performance of events in the *SPIN* operating system.

1 Introduction

SPIN is an extensible operating system that allows applications to define customized services to improve their performance, correctness or simplicity [Bershad et al. 95]. Applications dynamically add code to an executing system to provide new services, and they replace or augment old code to change existing ones. For example, an application may provide a new in-kernel file system, replace an existing paging policy, or add compression to network protocols. The system relies on a combination of compile-time, link-time, and run-time facilities to deliver good performance while providing the safety properties generally expected of a modern operating system.

This research was sponsored by the Advanced Research Projects Agency, and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship.

The key to the system's architecture is a mechanism that integrates applications' extensions with the system. *SPIN* uses *events* to provide such a binding mechanism. In an event-based system, components announce some system occurrence by explicitly *raising* an event of a particular name. Other parties, interested in learning of the occurrence, register *event handlers* which execute in response to a raised event. Events are generally recognized as an effective technique for implementing loosely-coupled, flexible systems in which relationships between code components must be dynamically established [Reiss 90, Sullivan & Notkin 92].

SPIN's use of events as its integration mechanism is novel in that interaction between components is entirely event-based down to the level of procedure call. Every procedure name in the system potentially represents an event, every procedure invocation corresponds to the raising of an event, and every procedure executing corresponds to the handling of an event. Consequently, all relationships between code components in the system are subject to change simply by changing the set of handlers associated with any given event.

The system provides an *event dispatcher* that oversees event-based communication. The dispatcher is responsible for enabling services such as conditional execution, multicast, asynchrony, and access control. In order to achieve good performance, the dispatcher is bypassed entirely to eliminate overhead in the common case of a single handler per event, and relies on runtime code generation to offer low-overhead performance that scales well as more handlers are installed for a given event.

1.1 Motivation

The kernel of the *SPIN* operating system defines only a few low-level services, such as device access, dynamic linking, and events – the subject of this paper. All other services, such as user-space

threads and virtual memory, are provided as extensions which are dynamically bound into the kernel as needed by applications. Consequently, the system's extensibility mechanism (events) must serve a broad range of purposes, including transparent interposition [Jones 93], multicast [Heidemann & Popek 94], filtering [Ritchie 84], and conditional execution [Mogul et al. 87]. Moreover, the system must be responsive to rapid and frequent changes in configuration, since extensions may come and go with the frequency of individual applications.

Our goal in defining *SPIN*'s extension services has been to provide the system and extension programmer with a set of interfaces that enable the system to be changed easily, transparently, safely, and efficiently. By *ease*, we mean that the system should lend itself to programmers making small changes with little programming overhead. By *transparent*, we mean that global effects should be achievable through local modifications. By *safe*, we mean that restrictions must exist to ensure that neither accidental nor malicious influences damage the system's integrity. Finally, by *efficient*, we mean that the runtime cost of potential and actual modifications to the system's structure should be small enough to ensure that programmers not be motivated to seek out alternative techniques for implementing change.

1.2 Related work

All software systems are extensible in one way or another, but it is the extension model and its implementation that determine the applicability and effectiveness of extensibility. These two properties are jointly influenced by the set of services that the model implies, and the performance characteristics of those services. A rich set of services enables a variety of interaction styles between the kernel and extensions, and between extensions themselves. Good performance enables that interaction to be freely applied without concern for execution overhead, while poor performance inhibits new interactions. For example, cross-address space interprocess communication has been commonly used in many systems to extend a system's functionality [Jones 93, Patience 93, Black et al. 92, Vahdat et al. 94]. Unfortunately, the high cost of protecting system code from extension code has generally limited the utility of this approach to coarse grain system interfaces. Some systems have relied on fast kernel upcalls to create specialized system services with lower kernel overhead [Thekkath & Levy 94, Engler et al. 95]. However, this approach does not lend itself well to extending shared services since all communication must be

channeled through the upcall mechanism.

Some systems have used dynamic linking to address the performance problems of cross-address space IPC [Orr et al. 93, Banerji & Cohn 94]. Dynamic linking tightly couples clients to a particular service implementation and does not provide for transparent routing of requests to alternate or supplementary services. As a result, dynamic linking does not allow multiple independent extensions to provide different parts of a service without explicit cooperation.

Events [Reiss 90, Sullivan & Notkin 92] have been used in several systems including network operating systems [Bhatti & Schlichting 95], windowing systems such as X11, software engineering environments [Cagan 90], extensible applications such as Emacs, database servers [Sybase 96], and the Macintosh operating system. Events support flexible composition by allowing an extension to be executed in response to interactions between components of the system. For example, in database systems, users can provide procedures that execute in response to certain database modification events, or *triggers* [Sybase 96]. Under windowing systems and the Macintosh operating system, events are used primarily to offer programmers a rudimentary concurrent programming interface, without forcing them to use threads. Unfortunately, traditional event systems have usually required a unique invocation syntax and protection model, and have not been as efficient as systems based on procedure call. As a result, events have been limited to specific services of an operating system and have not been applied to provide a system-wide extensibility mechanism.

Programming languages provide a number of approaches to extensibility that have been successfully applied in operating systems. For example, modularity [Parnas 72] is used in Oberon [Mossenbock 94], object-orientation [Goldberg & Robson 83] in Spring [Hamilton & Kougiouris 93], and reflection [Maes 87] in Apertos [Yokote et al. 89]. These approaches enable extensibility through the language mechanisms (for example inheritance) to modify language defined components (objects). However, programming languages have not typically provided support for multicast, transparent interpositioning, filtering, conditional execution, or access control, which are necessary services in an extensible operating system.

1.3 The rest of this paper

In the rest of this paper, we describe the design, implementation, and performance of events within

the *SPIN* operating system. In Section 2 we present the design of the system's extensibility mechanisms. In Section 3 we describe the implementation and performance of the event system. Finally, in Section 4 we conclude. Although this paper draws upon our experience with an extensible operating system, its ideas and mechanisms may be applicable to any software that admits dynamic restructuring [Sybase 96, Sun 95, Brockschmidt 94].

2 Events in *SPIN*

The *SPIN* kernel and its extensions are written in Modula-3, a modular, type-safe, ALGOL-like language. Extensions are incorporated into the system through a two-step process. First, the extension's code is dynamically linked into the operating system kernel [Sirer et al. 96]. The dynamic linker resolves all outstanding unresolved references in the extension code against a collection of interfaces explicitly exported by the system. This first phase is sufficient to allow the extension code to call into exported services, but does not provide a means for pre-existing code to interact with the extension code. That occurs during the second phase in which the extension registers handlers with events. The events are defined in interfaces against which the extension has been linked. Once installed, other extensions may link against the extension's exported interfaces and install handlers on the events it exports.

2.1 Defining events

Programs in Modula-3 are written as a composition of *modules* that communicate by procedure calls through *interfaces*. A procedure call can be viewed as one module's signal that another should execute some code. We exploit the basic relationship between procedure callers and implementations to define events. We treat a procedure call as an event raised by the caller and handled by one or more implementations. Because every procedure is implicitly an event, it is eligible for extension.

Events are described as Modula-3 procedure signatures, which has several implications. First, it preserves the "feel" of the language. Second, an event is a typed name, so it is protected from misuse and improper disclosure by the type system. For events declared in an interface, the event can be raised and in some cases handled by the modules that import that interface. An event not declared in an interface can be explicitly passed between modules as a typed procedure pointer. Finally, an event can be raised

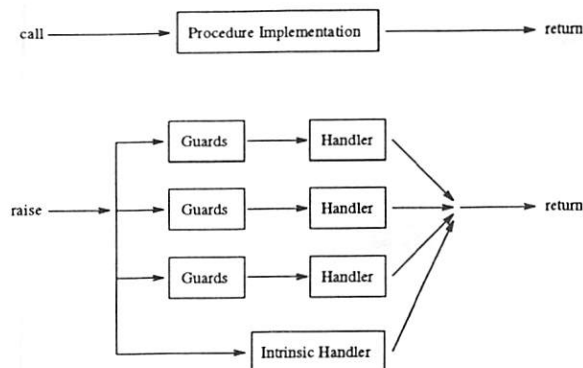


Figure 1: *Procedure call vs. event invocation.* A procedure call is directed to a single implementation. In contrast, an event is conditionally dispatched to any one of a number of implementations. The intrinsic handler is the procedure that has the same name as the event.

with arguments and return a value, allowing events to be parameterized and generate results.

A *handler* is a procedure that executes in response to an event. Handlers are dynamically added to or removed from an event, and any number of them can be installed at any particular time. The same handler can be installed many times on many events, and is invoked independently for each of the installations. Optionally, a handler may be installed with a *closure*, which is an opaque data reference. When the handler executes in response to an event, the closure specified at installation time is passed to the handler. This allows the same handler to be associated with multiple events.

The procedure having the same name as the event itself is its *intrinsic* handler, and it is invoked whenever the event is raised (unless the handler has been explicitly deregistered). A typical model for changing the implementation of a single procedure within a module is to deregister the intrinsic handler and then register an alternate one.

Each event handler can be associated with a set of *guard* predicates that filter out unwanted handler invocations. For example, an extension that is interested in handling page fault events for its data segment can define a guard that checks whether the faulting address is in that segment. The guards are specified outside of the handler so that the same handler can be installed multiple times with different guards, and so that additional guards can be added to further restrict when the handler can run.

Figure 1 illustrates the difference between conventional procedure invocation and event handling. A

procedure call is an invocation of the procedure's implementation through the procedure's name. An event raise is a conditional invocation of a collection of handlers installed on the event through the event's name. Of course, an event with only an intrinsic handler is identical (in semantics and implementation) to a procedure call.

```

(* Interface to trap handling *)
INTERFACE MachineTrap;

(* declaration of the MachineTrap.Syscall event *)
PROCEDURE Syscall(strand: Strand.T;
                  VAR ms: MachineCPU.SavedState);

END MachineTrap.

```

```

(* Implementation of the Mach extension *)
MODULE MachEmulator;

(* the syscall routine that
   handles Mach system calls *)
PROCEDURE Syscall(strand: Strand.T;
                  VAR ms: MachineCPU.SavedState) =
  BEGIN
    CASE ms.v0 OF
      ...
      | -65 => (* vm_allocate *)
        VMHandlers.vm_allocate(strand, ms);
      ...
    END;
  END Syscall;

FUNCTIONAL
PROCEDURE SyscallGuard(
  strand: Strand.T;
  VAR ms: MachineCPU.SavedState)
  : BOOLEAN =
  BEGIN
    RETURN IsMachTask(strand);
  END SyscallGuard;

(* Initialization *)
BEGIN
  (* installation of the syscall handler *)
  Dispatcher.InstallHandler(
    MachineTrap.Syscall,      (* event *)
    SyscallGuard,             (* guard *)
    Syscall);                 (* handler *)
END MachEmulator;

```

Figure 2: The Mach extension's system call routine installed as a handler on an event that announces a system call. The keyword "FUNCTIONAL" is described shortly.

2.2 An example

The implementation of system calls in *SPIN* demonstrates the use of events. The kernel provides no

native system call handling facilities. Instead, the MachineTrap module, which implements basic trap handling, exports an event Syscall through the MachineTrap interface. The event takes two arguments, the thread (an object of type Strand.T) in whose context the system call was made, and the saved state of that thread. When a system call trap happens, the machine dependent part of the kernel saves the state of the trapping thread, changes the state of the system to allow safe execution in the kernel context, and raises the MachineTrap.Syscall event. The dispatcher then evaluates the guards for each handler registered for this event, invoking the handlers for which the guards evaluate to true.

An extension providing its own system call service handles the MachineTrap.Syscall event as shown in Figure 2. This example, which shows an extension that emulates the Mach system call interface, illustrates several aspects of event handling. First, the event itself, Syscall, is defined as a typed procedure within the MachineTrap interface. Second, the extension code provides a handler for that event by registering a properly typed procedure through the dispatcher interface. The registration specifies the event name, any guard to be used, and the handler. In the example, the guard ensures that only system calls raised for threads executing as part of Mach tasks are handled by Syscall.¹

2.3 Evaluating guards

The dispatcher determines which handlers should be invoked in response to a raised event by evaluating their guards. Guards must be strictly functional, free of any side-effects. Since they describe predicates on the system's state, including any parameters passed to the guard, this restriction does not impose any significant computational restriction. If the processing of an event requires some side-effecting computation, then it may be encoded directly within the handler, and not within the guard. Lack of side effects is important because it prevents a guard from modifying or saving its arguments. This allows the dispatcher to reorder or short-circuit guard execution entirely in order to improve performance. In addition, it prevents arguments from being "leaked" out and stored away during an event invocation. A side-effect free procedure is declared as FUNCTIONAL within the source, which is verified by the compiler, and carried forward to execution time in

¹This example is somewhat contrived, since it shows the Syscall module limiting its own access to Mach system calls. In reality, additional guards are imposed to restrict visibility to events. We describe this facility in a later section.

the type information for the procedure.

Ordering handlers

Multiple handlers installed on the same event should be executed in an order that respects their dependencies. The dispatcher provides a deterministic mechanism for ordering handlers to avoid the need for explicit synchronization. Specifically, a set of ordering constraints can be associated with a handler. A handler installed with the First or Last constraint will be placed, respectively, at the beginning or the end of the handler list at the time it is installed. A handler can also be ordered relative to other handlers using a Before or After constraint. In addition, the dispatcher allows the ordering constraints associated with a given handler to be queried and dynamically changed.

Passing arguments

The dispatcher is responsible for passing arguments to guards and handlers. Generally, handlers and guards receive the arguments specified when an event is raised. If a handler or guard is installed with a closure, the closure is passed as an additional argument.

Handlers can be used as filters, wherein a single event can have multiple handlers, and handlers can modify the arguments seen by subsequent handlers. For example, an extension can provide the MS-DOS file name space over a UNIX file system by transparently converting file names from one standard to the other. Filters require that a handler be able to manipulate its arguments; consequently handlers explicitly installed as *filters* may specify some parameters as call-by-reference, rather than call-by-value. The filter can change these arguments and the dispatcher passes the new values to handlers and guards ordered after the filter.

Handling results

All events, even those that return values, may have zero, one, or many handlers. If only one handler fires, then the value returned from that handler is passed on to the event raiser, mimicking procedure call. In case no handler runs, a runtime exception is thrown at the point the event is raised, enabling the event raiser to recover. Alternatively, an event may have installed on it a *default handler*, which executes only when no other handler fires. In case several handlers run, it is necessary to determine which, if any, of the results should be returned in response to

the event raise. This is performed by a *result handler*, which is called separately for each result; it ultimately determines the final result of the invocation. For example, the system defines a VM.PageFault event, which is raised on any page fault. Its return value is a boolean indicating whether the page is accessible. If the page is inaccessible, the VM system crashes the application. The default handler for this event relies on a trusted default paging service provided by VM. The result handler for this event returns the logical-or of all the handler results. Previous structuring approaches have addressed the problem of handling multiple results in a similar fashion [Pardyak & Bershad 94, Cooper 85].

2.4 Safety issues

The fundamental property of the system's event model is that every procedure call can potentially be dispatched to any one of a number of handlers. In practice, though, this property is overly lax, as it would enable any extension to potentially undermine the behavior of any interface within the system. In the rest of this section, we describe the protection mechanisms in place which ensure that *SPIN*, as an extensible system, can be managed in a controlled fashion. First, though, we informally describe the typechecking rules governing event and handler specification.

Typechecking

During installation, the dispatcher checks the types of handlers and guards to guarantee the type safety of event interactions. Several rules govern the type checking of handler installation. The handler's argument types and return value must be the same as that of the event's. For the guard, argument types must be the same as the event's, but the return value must be a boolean (indicating whether or not the guard evaluates true). A procedure installed as a handler with a closure must take an additional first argument of some reference type, and the type of the associated closure must be a subtype of that reference type. Although the arguments to the event and handler must be the same, a handler acting as a filter is allowed to specify some of its parameters as "by reference" rather than "by value." To preserve the initial value of the parameter from the standpoint of the event raiser, the dispatcher first makes a copy of the arguments and passes to the filter a reference to the copy.

2.5 Access control

We rely on a set of static and dynamic access control mechanisms to ensure that events are not misused by extensions. There are several aspects to the issue of access control with respect to events, such as: who is allowed to raise or handle an event, which handlers are allowed to handle a particular occurrence of an event, and who can manipulate an event's default or result handlers.

Fundamentally, our approach is to assert that some set of functions have *authority* over an event, and that those functions may directly affect the way in which that event can be used by the system. The authority may in turn, rely on notions of identity, context, or passwords to further implement more specific notions of access control.

An event's authority could be defined in many different ways. We selected a strategy that was easy and efficient to implement, and could be easily described to programmers. The authority for an event is the module that defines the event's intrinsic handler. Recall that the intrinsic handler for an event is that procedure having the same name as the event. For example, if some module *M* implements a procedure *M.P*, then *M* (rather, all code within *M*) is the authority over event operations on *M.P*. This approach allows the programmer to locate control over an event with the code responsible for defining the event in the first place. In addition, authority can be easily checked using type information maintained by the kernel runtime.

An event's authority specifies an authorization procedure for operations that concern the event. The dispatcher calls back into the authorization procedure every time the set of handlers and guards associated with the event is manipulated. An authorizer is passed information describing the operation that is being requested, some context information describing the requestor, and, optionally, an opaque reference passed in by the requestor that can be used to bootstrap a richer authorization protocol such as one based on passwords. The authorizer evaluates the request and then allows or prevents the operation.

So that authority can be demonstrated, we added a new type to the language runtime that describes compilation units (interfaces and modules), as well as operations for obtaining instances of those types [Hsieh et al. 96]. The operations guarantee that the identity of a module can be obtained only inside of that module. Upon receipt of a module descriptor and an event, the dispatcher checks that the described module in fact defines the intrinsic handler

for the event, and, if so, considers the caller an authority.

The simplest form of authorization concerns dynamic linking. When a module requests that it be dynamically linked against some other module, that module's authorizer is consulted and the linkage is permitted or denied. Denial prevents the requester from accessing any of the symbols, and hence events, exported by any of the modules governed by the authorizer. This strategy is sufficient for preventing modules from inappropriately raising events defined by other modules.

Authorization also occurs whenever a module attempts to install a handler for an event. The dispatcher, upon the installer's request, first contacts the authorizer to ensure that the requestor and the handler are legitimate. At this point, the authorizer has several options. It can deny the request, returning an error to the installer. Or, it can allow the request, and possibly apply some execution property, such as ordering constraints, onto the handler to ensure that previously installed handlers continue to operate as expected. Lastly, the authorizer can *impose* an additional set of guards on the handler. These guards act as those described in the previous section – they must evaluate to true in order for the handler to execute.

Imposed guards limit access to an event by dynamically filtering them before they are delivered to a handler. Any number of guards can be imposed on a handler, and they can be added and removed dynamically. Figure 3 demonstrates the use of authorities and imposed guards for the MachineTrap.Syscall event described earlier. The main initialization routine of the MachineTrap module, as an authority over the event, provides the dispatcher with a procedure that will be called whenever any other module attempts to install a handler for the Syscall event. On installation, the dispatcher calls *AuthorizeSyscall*, which in turn imposes a guard on the event ensuring that only system calls for the installing thread's address space will be seen by the new handler. Any system call that occurs for threads not executing as part of the installing thread's address space will not be seen by the handler being installed.

The example shows that the policies and granularity for event filtering are under the control of an event's authority. Obviously, different authorities may impose different restrictions, depending on the kind of service that is being offered through the event. For example, our networking code imposes a guard that restricts an application's extension to receive packets only when the packets' destination is

```

MODULE MachineTrap;
...

(* the authorizer for the MachineTrap.Syscall event *)
PROCEDURE AuthorizeSyscall (
    binding : Dispatcher.Binding)
    : BOOLEAN =
BEGIN
    (* impose a guard to ensure that the extension only
       fires for the currently address space context *)
    Dispatcher.ImposeGuard(binding,
        ImposedSyscallGuard,
        GetCurrentAddressSpace());

    RETURN TRUE;
END AuthorizeSyscall;

(* imposed guard for handlers
   installed on the MachineTrap.Syscall event.
   validSpace is the address space for which our
   associated handler has access;
   strand is the thread context that invoked the
   system call;
   ms is the machine state at the time of
   the call. *)

PROCEDURE ImposedSyscallGuard (
    validSpace : REFANY;
    strand : Strand.T;
    VAR ms : MachineCPU.SavedState)
    : BOOLEAN =
BEGIN
    RETURN Space(strand) = validSpace;
END ImposedSyscallGuard;

(* Initialization.
   We define an authorizer (AuthorizeSyscall)
   over the Syscall event. We present a
   descriptor (THIS_MODULE()) which, at runtime,
   can be checked to ensure that MachineTrap.Syscall
   is intrinsically defined within this module. *)
BEGIN
    (* install the authorizer by demonstrating
       authority *)
    Dispatcher.InstallAuthorizerForEvent(
        AuthorizeSyscall,
        MachineTrap.Syscall,
        THIS_MODULE());
END MachineTrap;

```

Figure 3: The authorization code that protects the event announcing a system call (*MachineTrap.Syscall*). The *validSpace* argument passed into the imposed guard is a closure that was provided as the final argument to *Dispatcher.ImposeGuard* within *AuthorizeSyscall*.

for a port that had been previously assigned to the application [Fiuczynski & Bershad 96].

2.6 Denial of service

Access control mechanisms ensure that only approved handlers execute in response to published events, but do not address the problem of service denial by those handlers. Specifically, we must be sensitive to the problem of resource hoarding, which occurs whenever an extension acquires “too much” of some system resource as a consequence of using events.

Runaway handlers

A handler that never returns can prevent the event raiser and other handlers from making progress. This can occur when a module raises an event which is being handled by another module that can not be trusted to return in a timely fashion. In terms of a view on a layered system, this happens whenever control “flows up.” The kernel is itself preemptible, so a runaway handler only prevents the event raiser from making progress; other threads in the system continue to run.

We offer two solutions to the problem of runaway handlers; one preventative, but expensive, and the other corrective, but cheap. The expensive solution is to run asynchronously. An asynchronous event or handler detaches the execution of an event raiser from the handler. Either the event itself, or a specific handler, can be specified as *asynchronous*. If the event is asynchronous, then all handlers execute in a separate thread, and the event raiser proceeds without blocking. At times it may be advantageous to execute only some of the invoked handlers asynchronously, so the dispatcher allows any handler to be invoked asynchronously. For example, if a file system is extended with lazy replication, the original code should perform the write synchronously, but the replication can be done asynchronously.

Asynchronous execution incurs the additional expense of spawning off a new thread of control for each raised event, but is sufficient for procedures that are part of slow operations, and for which replies are not necessary. For example, our in-kernel UNIX server uses asynchronous events to implement a per-application system call tracer, and our virtual memory system uses asynchronous events for page-in requests.

The asynchrony of either the event or the handler is specified during handler installation. *SPIN* does not allow the use of asynchrony where it would vi-

olate the semantics of a procedure call. Since no immediate result is returned from an asynchronous event raise, an attempt to raise an event asynchronously that returns a result will raise an exception unless a default handler is installed. Because asynchronous threads execute on different stacks, arguments can not be passed by reference; they may be incidentally destroyed before they go out of scope. Therefore, it is illegal to define as asynchronous an event that takes an argument by reference, or to install an asynchronous handler on such an event.

When performance is critical, runaway handlers can be avoided by a corrective measure: termination. Handlers that execute beyond a certain time period (specified by the event's authority) are terminated. Of course, early termination introduces a variety of safety problems with regard to the type system and program correctness. For example, if a handler is terminated in the middle of allocating some memory, or updating a critical data structure, then the entire system could be left in an unsafe state.

We address the interaction between early termination and safety by restricting the installation of procedures that might be terminated to those that explicitly invite termination. Handlers willing to be terminated must be explicitly declared as EPHEMERAL within the source [Hsieh et al. 96]. The expectation is that an EPHEMERAL handler returns quickly, but, if not, will be terminated. An authorizer can determine whether or not a particular handler is in fact EPHEMERAL, and refuse installation if it is not. Program correctness can not be affected by the termination of an EPHEMERAL handler because, by definition, such termination is correct. Moreover, termination is localized to affect only EPHEMERAL handlers; the compiler ensures that EPHEMERAL procedures may only interact with other such procedures. Consequently, any subsystem can protect itself from early termination simply by exporting no EPHEMERAL handlers.

We tend to use EPHEMERAL handlers where synchronous interaction is required between low levels of the kernel and simple extension code. For example, our network interrupt handlers communicate incoming packets to extensions through handlers marked as EPHEMERAL. A terminated handler in this case simply causes a packet to be lost. As another example, extensions that manage user-space threads rely on EPHEMERAL handlers to save and restore thread state during context switches. Premature termination results in the termination of the user-space thread, which is followed by a termination of the user-space task itself.

Too many handlers

In theory, there can be no limit to the number of handlers or guards associated with any given event. In practice, though, events having more than one handler or guard consume some amount of kernel memory. Consequently, an extension could exhaust the system's memory by installing a large number of handlers on an event. Presently, *SPIN* denies additional installations when memory is low, relying on individual authorizers to locally enforce restrictions. We do not consider this to be a good long term solution, though, and realize that it is an instance of the more general problem of resource management given a shared pool. We are currently experimenting with different strategies for accounting and resource reclamation and hope to report on these in a future paper.

2.7 Summary

SPIN's event services provide a few mechanisms that allow the program to codify their intents about component relationships. These relationships are defined through explicit requests on a dispatcher service, and are exercised using the syntax of procedure invocation. Although events allow any relationship to be defined dynamically, most are in fact static, and most procedures are used as procedures (no guard, single handler), and not as events. Where more flexible compositions are required, though, the event mechanism we have described allows them to be forged "from a distance." Lastly, the event facilities include access control and protection services to guard against their misuse and abuse. It must be clear, though, that a certain degree of vigilance and paranoia is required when designing the components of an extensible system.

3 Implementation and performance

In this section we describe the implementation and performance of the event services discussed in this paper. The *SPIN* operating system and its extensions run on DEC Alpha Workstations and Intel X86 Personal Computers². The system is about two and a half years old, and has currently about 100 different extensions of various sizes and complexity, including six different file systems, two different operating system emulators, the Internet protocol suite,

²We present performance numbers only from the more mature Alpha version.

and a collection of integrated applications, including a distributed transaction system and a web server. Events are used to implement a number of critical system services, including system calls and emulation, threads and scheduling, networking, sockets, and tty services. With our Unix emulator running, the current version of the system installs handlers on 57 different events, with an average of 3.5 handlers per event.

In order to achieve good performance for events, the dispatcher is tightly coupled with the Modula-3 runtime. An event with only an intrinsic handler is dispatched directly as a procedure call (which, in fact, it is). For richer events, the dispatcher relies on direct access to the compiler-generated information to hide the event mechanisms behind the Modula-3 procedure syntax and semantics. For example, the dispatcher uses runtime type information to type-check procedures when they are installed as handlers and guards. It directly manipulates runtime data-structures such as module and interface descriptors, and method dispatch tables to ensure that events can be dispatched with the overhead of procedure call.

For events having guards, multiple handlers, or other extended semantics, the dispatcher replaces the implementation of the intrinsic procedure with one that performs the appropriate dispatch. The dispatcher maintains a separate list of handlers for each event. When a handler is added to or removed from an event, the list is regenerated and used for subsequent event dispatches. The dispatch itself is performed by a routine that iterates over the handler list and invokes handlers according to the specified event properties. A handler can be installed and removed from an event dynamically and without disrupting on-going interactions. The handler lists are updated atomically with respect to event dispatch by using a single memory access to replace the old list with the new one.

We use run-time code generation to build a specialized and optimized version of the dispatch routine. Our approach is similar to that taken in previous systems which have used run-time code generation for late-bound kernel code [Massalin & Pu 89]. We specialize the code to the number of arguments in each event, and unroll the dispatch loop to transform handler invocations from indirect procedure calls through a list of handlers to direct procedure calls. We also inline the code of small guards and handlers directly into the dispatch routine. Finally, we use peephole optimizations to improve the quality of the generated code.

The dispatcher itself consists of about 3400 lines

of Modula-3 code, and the run-time code generator and optimizer take another 7600 lines of Modula-3, C, and Alpha assembler code. The rest of the kernel is substantially larger, and includes the Modula-3 runtime libraries (30000 lines), device drivers and debugging support for the family of DEC Alpha workstations (340000 lines)³, and other core kernel services including dynamic linking, scheduling, and memory management (29000 lines).

3.1 Performance

We report on several aspects of event performance. Our measurements are intended to demonstrate that events have low overhead, and that the overhead scales well with the number of handlers installed on a particular event. Lastly, we show the impact of event management on overall application performance. We collected our measurements on DEC Alpha 133MHz AXP 3000/400 workstation, which is rated at 74 SPECint 92. We ran our experiments on *SPIN* V1.26 of September 1996.

Latency

Table 1 shows the latency of synchronous event handling as a function of the number of guards and handlers installed on an event. Guards compare a global variable to a constant and return true, and handlers return without performing any work. We measured latency in the case where guards and handlers execute out of line, and in the case where the dispatcher inlines them. In practice, inlining occurs only when there are relatively few handlers, and when they are relatively small. The table shows that dispatch overhead is primarily a function of the number of procedure calls that occur during each raised event. The simple case, where an event has only an intrinsic handler is implemented as a straight Modula-3 procedure call from the raiser to the handler, and performs as shown in the table. For a small number of handlers it is the same order of magnitude as a procedure call. When additional guards or handlers are installed, overhead grows linearly with the number of guards and handlers, with each incurring an overhead of roughly one procedure call. Asynchronous events, which have not been optimized, introduce an additional latency of between 38 and 90 μ secs per event raised. The additional time is spent creating the asynchronous thread. In terms of its impact on basic system services (microbenchmarks), we have

³We borrow all of this code directly from the manufacturer's source tree.

Number of arguments	Modula 3 procedure call (intrinsic)	Number of event handlers							
		1		5		10		50	
		no inline	inline	no inline	inline	no inline	inline	no inline	inline
0	0.10	0.37	0.23	1.18	0.41	2.15	0.63	11.69	2.48
1	0.13	0.39	0.24	1.25	0.45	2.32	0.72	11.51	2.87
5	0.14	0.97	0.42	1.61	1.55	2.88	1.32	14.45	5.65

Table 1: *The overhead of event dispatching on Alpha workstations. Most events with non-intrinsic handlers in SPIN have between one and ten handlers. Times for both inlined and non-inlined handlers and guards are shown. All times are in μ secs.*

measured event processing overhead to be on the order to 10–15% for operations such as system call and thread management.

Installation overhead

Each time a new handler is installed for an event, the dispatcher regenerates the data structures and code associated with that event. Consequently, the overhead to install n handlers is $O(n^2)$, although in practice this quadratic is not a serious problem. Most events have few handlers on them, and even with many handlers, overhead is tolerable. The time to install a single handler is about 150 μ secs, whereas to install 100 handlers on the same event takes about 30 milliseconds. We've not yet made any effort to reduce this overhead. Intrinsic handlers – that is, most procedures in the system – are defined without any runtime overhead. Nevertheless, we may ultimately, need to rely on a more incremental (and economical) approach to installation.

3.2 End-to-end effects

We ran two more experiments to better understand the end-to-end effects of event management on overall performance. In the first, we show how networking latencies are affected by flipping packets back and forth between two machines. In the second, we calculate a breakdown of where time is spent in the system when displaying a Postscript image using X11 on *SPIN*.

Networking

SPIN's networking support is implemented in terms of events and guards [Fiuczynski & Bershad 96]. Guards are used to filter packets from the network by discriminating on fields in the protocol header (e.g., guards may discriminate on the UDP or TCP port destination field). The round-trip latency of a protocol reflects the overhead induced by the protocol as

1 guard	5 guards	10 guards	50 guards
475	481	487	530

Table 2: *Network UDP roundtrip time as a function of the number of guards installed on a packet event. Only one guard evaluates to true. Times are in μ secs.*

it transfers bytes from sender to receiver and back to the sender. Table 2 shows the round-trip latency for small (8 byte) UDP/IP messages between applications running on a pair of AXP 3000/400 machines running *SPIN* and connected by a 10Mb/s Ethernet. To highlight the overhead of event handling, we install additional guards on the `Udp.PacketArrived` event that evaluate to false. That is, the experiment has one active endpoint and many inactive ones, yet all guards are evaluated for each packet that arrives from the network.

The measurements show that each additional guard introduces an overhead of about one μ sec to the round-trip latency. Note that we do not presently reorder guard evaluation in order to accommodate packet trains, as is done by many packet filter implementations. In addition, we presently do not optimize the guard decision tree, which would be effective for the port comparison required by this example. We are currently working on a strategy by which this type of guard optimization can be easily expressed.

Application performance

In order to assess the impact of event handling on overall performance, we measured the *SPIN* kernel running Digital's X11 server while displaying a Postscript version of this paper. The Postscript was being processed on another machine running ghostview, an X11 Postscript previewer, and the page images were sent over the network to the *SPIN*

Event name	raised	time	handlers	guards
Ether.PacketArrived	2536	0.03	4	3
Ip.PacketArrived	2529	0.02	6	5
Udp.PacketArrived	24	0.00	6	5
Tcp.PacketArrived	2505	0.01	2	1
OsFNet.DelTcpPortHandler	3	0.00	1	0
OsFNet.AddTcpPortHandler	3	0.00	1	0
MachineTrap.Syscall	3976	0.03	3	2
Strand.Run	7936	0.03	4	3
Events.EventNotify	595	0.00	2	2

Table 3: Major events raised while previewing a document. Most of the events are related to protocol processing. Strand.Run occurs during each scheduling operation, and Event.EventNotify is raised by our implementation of the Unix `select` system call. The “raised,” “time,” “handlers,” and “guards” columns refer to the number of times a given event was raised, the total time in seconds spent handling the event, and the number of handlers and guards installed on the event.

machine.⁴ This setup stresses several kernel extensions including our Digital UNIX emulator supporting X11 and our TCP/IP networking stack [Fiuczynski & Bershad 96]. We instrumented the kernel and extension code to generate call graph information with counts and elapsed times. Table 3 reports on the kind and number of the most commonly occurring events during the system’s execution.

The total time required to preview the document was 23.5 seconds. Much of the time (12.52 seconds) was spent idling, 4.2 seconds were spent processing in the X11 server, and 6.8 seconds were spent executing in the kernel. Of that kernel time, 0.12 seconds were spent raising and dispatching events, or roughly .5% of total execution time, 1% of total CPU time, and 1.7% of total kernel time. These numbers are pessimistic because of dilation effects due to profiling, which tend to exaggerate the impact of fast routines (such as handlers and guards) on overall performance.

4 Conclusions

We have described the design and implementation of an event mechanism for an extensible operating system. In order to achieve good performance, we have designed the event mechanism “close” to the programming language, allowing us to leverage off of fast default paths, while still providing richer composition semantics where necessary. Our performance measurements demonstrate that event pro-

⁴ SPIN’s UNIX emulator still has trouble with a few UNIX programs. Ghostview is one of them.

cessing overhead is small, both in an absolute sense, where it is close to procedure call latency, and in an relative sense, where it represents only a small fraction of service and overall execution time.

More information on the SPIN project is available at <http://www-spin.cs.washington.edu>, an Alpha workstation running SPIN with a WEB server extension. Our home page also includes pointers to pages that document the dispatcher interfaces described in this paper.

Acknowledgments

The work described in this paper is part of a large operating system effort at the University of Washington. None of what we have described here could have been done without the input, assistance, and feedback of all the other members of the SPIN group, especially including David Becker, Marc Fiuczynski, Charlie Garrett, and Yasushi Saito. In particular, we owe special thanks to Stefan Savage and Emin Gün Sirer who contributed to the initial design of the event system. Wilson Hsieh helped design and implement many of the language extensions used by the dispatcher. Jan Sanislo implemented the system’s support services that enable us to run X11 on top of SPIN. Dylan McNamee, Gene Morgan, Sandy Morgan and Stefan Savage all helped with earlier drafts of this paper. Finally, we owe special thanks to John Wilkes, our shepherd, for his guidance and patience.

References

- [Banerji & Cohn 94] A. Banerji and D. L. Cohn. Protected Shared Libraries. Technical Report 37, University of Notre Dame, 1994.
- [Bershad et al. 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995.
- [Bhatti & Schlichting 95] N. T. Bhatti and R. D. Schlichting. A System For Constructing Configurable High-Level Protocols. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, August 1995.
- [Black et al. 92] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, WA, April 1992.

- [Brockschmidt 94] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
- [Cagan 90] M. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, June 1990.
- [Cooper 85] E. C. Cooper. Replicated Distributed Programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 314–324, December 1985.
- [Engler et al. 95] D. R. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995.
- [Fiuczynski & Bershad 96] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the 1996 Winter USENIX Conference*, January 1996.
- [Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language And Its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.
- [Hamilton & Kougiouris 93] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.
- [Heidemann & Popek 94] J. Heidemann and G. Popek. File-System Development with Stackable Layers. *Communications of the ACM*, 12(1):58–89, February 1994.
- [Hsieh et al. 96] W. Hsieh, M. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. Bershad. Language Support for Extensible Systems. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, pages 127–133, February 1996.
- [Jones 93] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Call Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC, December 1993.
- [Maes 87] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications*, pages 147–155, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.
- [Massalin & Pu 89] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park, AZ, December 1989.
- [Mogul et al. 87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.
- [Mossenbock 94] H. Mossenbock. Extensibility in the Oberon System. *Nordic Journal of Computing*, 1(1):77–93, February 1994.
- [Orr et al. 93] D. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the 1993 Winter USENIX Conference*, June 1993.
- [Pardiyak & Bershad 94] P. Pardiyak and B. Bershad. A Group Structuring Mechanism for a Distributed Object-oriented Language. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 312–219, Poznan, Poland, June 1994.
- [Parnas 72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.
- [Patience 93] S. Patience. Redirecting Systems Calls in Mach 3.0, An Alternative to the Emulator. In *Proceedings of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.
- [Reiss 90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, July 1990.
- [Ritchie 84] D. M. Ritchie. A Stream Input-Output System. *Bell Labs Technical Journal*, 63(8, Part 2):1897–1910, October 1984.
- [Sirer et al. 96] E. Sirer, M. Fiuczynski, P. Pardiyak, and B. Bershad. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, pages 141–148, February 1996.
- [Sullivan & Notkin 92] K. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.
- [Sun 95] Sun Microsystems. *The Java™ Language Specification*, 1.0 beta edition, October 1995.
- [Sybase 96] Sybase. Sybase SQL Server 11. Technical report, Sybase, Inc., August 1996. <http://www.sybase.com/products/system11/sqlsrv11.html>.
- [Thekkath & Levy 94] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 145–156, San Jose, CA, October 1994.
- [Vahdat et al. 94] A. Vahdat, P. Ghormley, and T. Anderson. Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report UCB CS-94-842, University of California, Berkeley, December 1994.
- [Yokote et al. 89] Y. Yokote, F. Teraoka, and M. Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In S. Cook, editor, *Proceedings of the 1989 European Conference on Object Oriented Programming*, pages 89–106, Nottingham, July 1989.

Dealing With Disaster: Surviving Misbehaved Kernel Extensions

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, Keith A. Smith
Harvard University

Abstract

Today's extensible operating systems allow applications to modify kernel behavior by providing mechanisms for application code to run in the kernel address space. The advantage of this approach is that it provides improved application flexibility and performance; the disadvantage is that buggy or malicious code can jeopardize the integrity of the kernel. It has been demonstrated that it is feasible to use safe languages, software fault isolation, or virtual memory protection to safeguard the main kernel. However, such protection mechanisms do not address the full range of problems, such as resource hoarding, that can arise when application code is introduced into the kernel.

In this paper, we present an analysis of extension mechanisms in the VINO kernel. VINO uses software fault isolation as its safety mechanism and a lightweight transaction system to cope with resource-hoarding. We explain how these two mechanisms are sufficient to protect against a large class of errant or malicious extensions, and we quantify the overhead that this protection introduces.

We find that while the overhead of these techniques is high relative to the cost of the extensions themselves, it is low relative to the benefits that extensibility brings.

1 Introduction

Many of today's research operating systems share the goal of providing applications with a richer and more powerful interface to kernel functionality. There are several approaches undergoing exploration and analysis today. The Scout system [9] supports static specialization: Scout administrators can run a kernel that has been specialized for a particular workload on a particular machine. By targeting a particular kernel for a particular workload, Scout can take advantage of advanced compiler optimization techniques, efficient kernel code paths, and a smaller system footprint. The extensible systems, such as SPIN [4] and VINO [15] allow applications to download code into the kernel to provide improved functionality and performance. Synthetix [13] provides improved flexibility and functionality by iden-

tifying commonly executed paths and producing optimized versions of them, but does not allow applications to modify or extend the kernel.

In this paper, we concentrate on the class of extensible systems. An extensible system is one that permits clients to modify the behavior of a shared server by loading client-specific extension code into the server. Such extensibility is useful in a wide range of systems. Database clients might extend their server by loading code into it to support new data types [8]. In a traditional operating system, user applications can exploit extensibility to customize the policies and functionality implemented by the kernel (e.g., the eviction policy for the file cache, or the delivery order for signals and other asynchronous events). Finally, in a microkernel operating system, the same extensions that are of interest in a traditional operating system can be loaded into the relevant system servers rather than the operating system kernel. Although we focus on extensibility in the context of a monolithic operating system kernel, the issues and technology discussed in this paper are relevant to these other classes of extensible systems.

An extensible operating system supports the downloading of application extensions, or *grafts*, into the kernel. Allowing applications to load code into the kernel spells immediate disaster unless the kernel is well-protected against buggy or malicious grafts. Safeguarding the kernel from errant grafts requires two different forms of protection. First, the kernel must guarantee that grafts do not misuse memory by reading inappropriate data (e.g., device registers or another user's data), writing inappropriate data, or executing bad instructions. This problem has been addressed by the use of safe languages such as Modula-3 [11], as used by SPIN, software fault isolation [20], as used by VINO, or virtual memory address domains, as used by Mach [1]. Second, the kernel must ensure that grafts do not consume resources to the extent that they jeopardize the acceptable performance of the kernel and other applications. This problem has been less well researched and is the topic of this paper.

In Section 2, we discuss the different ways that grafts can inadvertently or intentionally jeopardize system integrity. In Section 3, we discuss VINO's graft architecture and how it addresses the issues raised in Section 2. In Section 4, we quantify the cost of VINO's mechanisms. In Section 5, we present related work. In

This work was sponsored in part by grants from Sun Microsystems Laboratories, Bellcore, the Sloan Foundation, and the National Science Foundation.

Section 6, we discuss the lessons we have learned in building VINO, and we conclude in Section 7.

2 How Grafts Misbehave

Our model of grafts is that they are similar to regular processes that run inside the kernel. SFI is used instead of the traditional VM mechanisms to prevent illegal data accesses. Each graft receives its own heap and stack, and when a graft changes kernel state (e.g., by opening a file), the kernel records the fact so that any such modifications can be undone if the graft misbehaves. In a trusting world, these precautions are sufficient to avoid or cope with innocent errors such as access through an invalid pointer. However, in an untrusting world, kernel extensions might be malicious, seeking to destroy system integrity, performance, or security. In building an extensible system, we strive to prevent grafts from causing more damage than can be caused by a maliciously written user-level program. Therefore, we consider it unacceptable for a bug in a graft to crash the kernel, while it might be tolerable for a graft to loop infinitely, so long as it consumes only as much processing time as would a user-level program with the same infinite loop.

There are three reasons why a maliciously written graft is more dangerous than a maliciously written process. The first is that grafts run in supervisor mode. If no special care is taken, grafts have the potential to wreak havoc with the kernel. The second is that grafts are given access to a more powerful interface. Though still restricted, grafts have access to more kernel entry points than ordinary processes, including some of the kernel's synchronization points, providing grafts with simple and effective ways to sabotage the system. The more restrictive the graft interface, the easier it is to protect against malice, but the more limited the functionality of the grafts. It is a continuing struggle to determine the correct balance between expressive power and simplicity in designing a safe graft interface. The final reason that a graft can be more dangerous than a process is that once a graft is installed, the system relies on the correct operation of the graft to provide system services, and the graft's actions can potentially affect all the processes on the system. This problem is shared by other extension models, such as upcalls. Depending on the function that a graft or user-level server provides, a malicious one can prevent the system from making progress or can cause it to crash.

We have identified five classes of misbehavior that grafts might exhibit. Each is described below. We use the classifications to derive nine guiding principles for the construction of a stable, extensible operating system.

2.1 Illegal Data Access

Because grafts run in supervisor mode, we cannot use conventional virtual memory mechanisms to prevent grafts from making illegal memory accesses. Safe languages and software fault isolation provide mechanisms for limiting the data accessible to a graft. We must also provide a mechanism that allows the kernel to determine whether a graft has been processed or compiled by such a tool.

It is not sufficient to prevent a graft from accessing information to which it is not entitled; we must also ensure that a graft cannot execute a kernel function that can provide the graft with information to which it is not entitled. This means that any interface that returns actual data to its caller (as opposed to meta-data) cannot be called by a graft. In general, the kernel can pass meta-data (such as buffer headers) freely to grafts, so long as the data to which they refer (e.g., the actual data buffers) are protected.

2.2 Resource Hoarding

Grafts can consume system resources. They can attempt to loop infinitely, allocate excessive amounts of memory, or flood the network with packets. Because the interface given to grafts is more powerful than that given to user-level code, there is the potential for grafts to cause more serious damage. For example, if grafts are allowed to acquire kernel locks, they can block kernel progress more effectively than a process that is unable to directly acquire locks. Similarly, if grafts can consume kernel resources, such as physical memory, and hold them over long intervals, resource contention and starvation can cause significant problems.

Consider this malicious code fragment:

```
lock(resourceA);  
while (1);
```

If resourceA is highly contested, then we cannot let the graft hold it arbitrarily long. In fact, a graft cannot be allowed to hold any limited kernel resource for an arbitrarily long period of time.

We cannot rely on a static check to prevent grafts from hoarding resources. Therefore, we must be able to *preempt*, and, if necessary, *terminate* the thread from which a graft is called. If we terminate the thread, we undo any kernel state changes that have been made, releasing any resources held by the thread and allowing the rest of the system to make forward progress.

Through preemption and scheduling we can prevent a graft from taking more than its share of a resource. Even a graft with an infinite loop gets no more CPU time than would a user-level process with the same infinite loop.

1. Grafts must be preemptible (§2.2).
2. Grafts cannot hold kernel locks or limited kernel resources for excessive periods of time (§2.2).
3. Grafts cannot access memory to which they have not been granted permission (§2.1).
4. Grafts cannot call functions that alter or return data that the graft is not allowed to access (§2.3).
5. Grafts cannot replace restricted kernel functions (§2.3).
6. The kernel must not execute grafts that are not known to be safe (§2.1, §2.3).
7. Grafts must not call functions to which they have not been granted access (§2.1).
8. Malicious grafts can only affect applications that have agreed to use them (§2.4, §2.5).
9. The kernel must be able to make progress even with a faulty graft in its path (§2.2, §2.4, §2.5).

Table 1. Rules for Grafting. Based on the ways in which grafts might corrupt the kernel, we derive these rules for creating a safe, stable extensible kernel. We include the numbers of the sections that imply each rule.

2.3 Attempting to Use Incorrect Interfaces

There are certain parts of the kernel that cannot be extended for a single application. For example, a single application, running as a normal user, cannot be allowed to replace a global kernel policy. If it could, the application could take over the system by downloading a highly biased scheduler. Such global graft points must be accessible only to privileged users (users who, in a conventional system, would be allowed to halt the system, install new drivers, build a new kernel, etc.). Additionally, the kernel must somehow verify that the downloaded graft has been properly protected (e.g., compiled with the correct compiler). Finally, we must limit the functions that are callable from grafts. As pointed out in Section 2.1, grafts should not be allowed to call functions that return private data. Additionally, grafts should not be able to call functions that change kernel state in an unrecoverable fashion; a graft should not be able to call shutdown().

2.4 Antisocial Behavior

Our next category of graft misbehavior arises from grafts that simply do not do what they have agreed to do. Consider a scheduling graft used by a collection of clients and a server. Assume that the graft always selects the same process to run. This scheduling discipline penalizes the members of the scheduling group, but has no adverse effect on processes that use the normal scheduling discipline. We find this model of behavior acceptable, applying Cao's principle for defining an acceptable allocation policy: the selection of an application specific policy should not adversely affect other applications [5]. The task of the kernel is to prevent grafts from damaging the integrity of the kernel. We interpret this to mean crashing the kernel, corrupting data, or interfering with processes that have *not* agreed to use the graft.

2.5 Covert Denial of Service

A graft can attempt a denial of service attack, by taking advantage of the fact that the system is relying on its correct execution to make forward progress. A page replacement graft is such an example. When a page is about to be evicted, the page daemon calls the graft so that the graft can present an alternate page to replace. If the graft never returns, the page daemon cannot make forward progress causing the system to eventually run out of free pages. Therefore, it is essential to provide some mechanism by which we can detect such a throttling state and return the system to a state where it can make forward progress.

2.6 Summary

Table 1 summarizes the restrictions that we must impose on grafts if we are to protect the kernel. In the next section, we discuss how VINO addresses each of these restrictions.

3 The VINO Grafting Architecture

VINO is an extensible operating system designed to provide resource-intensive applications greater control over resource management. VINO supports the downloading of kernel extensions, which are written in C++ and protected using software fault isolation. To facilitate graceful recovery from an extension failure, VINO runs each invocation of an extension in the context of a *transaction*. If the invocation fails or must be aborted (e.g., because it is monopolizing resources), the transaction mechanism undoes all actions taken by the invocation of the extension.

The VINO kernel is constructed from a collection of objects and consists of an inner kernel and a set of resources. VINO provides two different modes of extensibility. First, a process can replace the implementation of a member function (method) on an object; this type of extension is used to override default policies, such as cache replacement or read-ahead.

Second, a process can register a handler for a given event in the kernel (e.g., the establishment of a connection on a particular TCP port). Extensions of this type are used to construct new kernel-based services such as HTTP and NFS servers.

VINO runs on Intel's x86 processors. The machine-independent parts of VINO consist of entirely new code. Following traditional engineering practices, we have encapsulated all of the machine dependent parts of the kernel behind a standardized interface. Beneath this interface we use the machine dependent portions of NetBSD 1.0—locore, the pmap module, and the device drivers. By using the machine dependent code from a pre-existing system, which itself runs on a variety of platforms, we hope to simplify the task of porting VINO to other architectures.

In this section, we discuss the VINO kernel transaction mechanism, which is used to allow the kernel to recover from misbehavior by extensions, and how the kernel determines when to abort an extension invocation. We then describe our software fault isolation tool and dynamic linker, and give examples of the two types of grafting.

3.1 Kernel Transaction Support

We encapsulate each graft invocation in a transaction to allow us to spontaneously abort a graft and clean up its state. When a function is grafted into the kernel a small wrapper function is interposed; the wrapper begins a transaction for the graft invocation and then calls the grafted function. When the grafted function returns, the wrapper commits the transaction.

The transaction support necessary for grafts is simpler than a conventional data manager's transaction mechanism. The single goal of graft transactions is to provide a means for backing out changes made by faulty grafts. Therefore, the transaction system does not need to handle permanent data, so its log need only be transient, and it never has to "redo" operations; it only has to undo them¹. Therefore, of the four "ACID" properties typically associated with transactions (atomicity, consistency, isolation, and durability), we need only provide the first three. However, because graft functions may indirectly invoke other grafts, we found it necessary to include support for nested transactions. In this manner, any graft can abort without aborting its calling graft.

All graft transactions are managed by the default VINO transaction manager. When a transaction is

initiated the manager allocates a transaction object that is associated with the thread that invoked the graft. The VINO transaction manager uses two-phase locking and an in-memory undo call stack. Because the kernel is preemptible, it must acquire locks on all resources being accessed or modified. In the non-transaction case these locks are released as soon as a thread is done manipulating the resource. When the currently running thread has a transaction associated with it, lock release is delayed until commit or abort.

Modifications to permanent kernel state are encapsulated in accessor functions (i.e. a grafted function cannot directly manipulate kernel data; it must go through data accessor functions). Each such accessor function that can be called from a grafted function has an associated undo function. Whenever an accessor function is called, if there is a transaction associated with the currently running thread, the corresponding undo operation is pushed onto the transaction's undo call stack². If a transaction aborts, the transaction manager invokes each undo operation on the undo call stack, and returns a transaction abort error to the graft stub, which then calls the default function (i.e., the function that was replaced by the graft).

When a non-nested transaction commits, the locks are released, the undo call stack and transaction object are freed, and execution continues normally. When a nested transaction commits, its undo call stack and locks are merged with those of its parent. Although different in implementation, graft transactions are similar in concept to the volatile transactions used in the Quicksilver system [7].

3.2 When to Abort Graft Transactions

Transactions provide the mechanism by which we can abort resource intensive grafts, but we still need a policy to determine when to abort a graft. Grafts are allowed to run so long as they do not interfere with the behavior of other processes. For the purpose of discussing resource hoarding, we can divide the various system resources into two categories. For some resources, we are primarily concerned that a graft does not hold the resource for too long, thus becoming a bottleneck to all other threads that need the resource. We call these *time-constrained resources*. With other resources, such as memory, we wish to ensure that a graft does not use too much of the resource. We call these *quantity-constrained resources*. In VINO, we use a different technique to enforce limits on each of these types of resource.

1. Note that conventional transaction semantics can be provided by creating a new instance of our transaction manager and replacing the transient log manager with a permanent one.

2. Our current implementation requires that this code be added by hand. This could clearly be automated with a preprocessor and appropriate source-code decoration.

Consider the case of locks as an example of a time-constrained resource. If a graft holds a lock that no other thread requests, then continuing to hold that lock does not affect the rest of the system. Conversely, if other threads do request the lock, then the graft is potentially degrading system performance. Therefore, with every lockable resource, we associate a time-out value that indicates how long a lock can be held on that object during periods of contention. This time-out based locking also provides an implicit mechanism for breaking deadlocks. Because resource requirements vary tremendously, reasonable time-out intervals must be determined (experimentally) on a per-resource-type basis. For example, a page may be locked for tens of milliseconds during I/O while a free space bitmap should be locked for only a few hundreds of instructions while it is being traversed.

When a request for a lock blocks, the waiting thread schedules a time-out whose duration is based on the resource being requested. If the time-out on a lock expires, and the lock is held by a thread that is executing a transaction, we abort that transaction. Note that we abort the transaction even if the lock was acquired before the graft was invoked. In such a case, the graft will return to the invoking code which presumably will release the lock(s) in a timely manner.

To enforce limits on quantity-constrained resources, we use the same mechanisms for grafts that we use for user-level threads and processes. Each thread in VINO has a set of resource limits associated with it. These limits constrain the amounts of various resources (e.g., memory) that the thread may consume. When a graft is installed, it initially has limits of zero (i.e., it cannot allocate any resources). The installing thread may transfer arbitrary amounts from its own limits to the newly installed graft, or the thread can request that all of the graft's allocation requests be "billed" against the installing thread's own limits. If multiple processes wish to pool resources (e.g., a collection of database clients and servers may wish to pool their wired memory resources to create a shared buffer pool), they can each delegate their resource rights to the graft, in a manner analogous to ticket delegation in lottery scheduling [21].

When a thread invokes a grafted function in the kernel, the thread's resource limits are replaced by those associated with the graft. Thus, the same mechanisms that prevent processes from exceeding resource limits are automatically applied to grafts. When the process would normally be denied requests for new resources, the graft's requests also fail.

3.3 Graft Code Safety

As stated above, grafts are protected through the use of software fault isolation [20]. The overhead of software fault isolation has been shown to range from 5% to 200%, depending on the application. We developed an SFI tool, MiSFIT, for this purpose [17]. At compilation time MiSFIT inserts instructions to protect loads and stores. Code is added to force the target address to fall within the range of memory allocated to the graft. The cost of this protection is two to five cycles per load or store.

To protect function calls, VINO kernel developers maintain a list of *graft-callable* functions. Only functions on this list may be called from grafts. Direct function calls are checked when grafts are dynamically linked into the kernel; the function is looked up in the graft-callable list; if the target function is not on the list, the graft is not loaded into the system.

Indirect function calls (e.g., C++ virtual function calls) are checked at run-time by looking up the address of the target function in a hash table containing the addresses of all graft-callable functions. If the target function is not on the list, the graft's transaction is aborted. In general, the cost of probing a hash table depends on the contents of the table and the key being probed for. Through the use of a sparse open hash table we find our average cost is ten to fifteen cycles per indirect function call.

Graft-callable kernel routines must perform the same type of argument checking and verification that system calls do. A graft is run with the user identity of the process that installs it; graft-callable functions are responsible for checking that the user has been granted access to files, memory, and devices that the graft attempts to use. In this way the protection domain in which the graft runs is (at least in theory) the same as the protection domain of the process that installed the graft.

VINO must ensure that code loaded into the kernel has been processed by MiSFIT. MiSFIT computes a cryptographic digital signature of the graft and stores it with the compiled code. When VINO loads a graft it recomputes the checksum and compares it with the saved copy. If the two do not match the graft is not loaded. Tools that perform this type of code signing are commercially available [10].

3.4 Function Graft Example

Once a graft has been compiled, processed by MiSFIT, and assembled, it is ready to be grafted into the running system. To install a graft, an application must first obtain a handle for the *graft point*. This is accomplished by looking up the graft point in a kernel-maintained graft namespace. The name is composed of the object to

```

file_o *db;
graftpoint_handle_o *gp;

db = file_o::open("db", "r");
gp = graft_namespaces->lookup(db,
                               "readahead");
gp->replace("my_readahead.o");

```

Figure 1. Function graft example. The database server is replacing the kernel's default file read-ahead function with an application-specific version. The server looks up a handle for the read-ahead function of the database in the graft namespace. It then installs the new read-ahead function. Although the function calls shown here are to C++ member functions, they invoke VINO system calls.

be grafted (e.g., the open file) and the name of the function to be replaced (e.g., "read-ahead"). The graft point handle provides a *replace* method that is used to instruct the kernel to replace the function at the graft point with the new function. Figure 1 shows an example.

This interface enables the replacement of a single member function for a given object. The list of functions that can be grafted on each class is specified by the class designer; some classes may not allow any of their functions to be grafted; others may allow all functions to be grafted.

3.5 Event Graft Example

The interface discussed above is suitable for modifying the behavior of a single object. However, an application may want to drop an entire service into the kernel, such as an HTTP server [4], an NFS server, or a database server. Our *event graft* model is based on the idea that these services are typically, if not always, designed to respond to a stream of incoming external events. Each of these servers receives a request, processes it, and sends a response. We model servers as handlers for events, where each request is viewed as an event. We extend our definition of function graft points, introduced above, to encompass these events: event graft points correspond to the external events to which a service responds.

Along with the replacement of a graft function shown above, we also permit the *addition* of a new graft function to a graft point. Rather than replace an existing function, the grafted function will be called in addition to any other functions added to the graft point. We provide an interface for applications to specify the order in which grafted functions are called.

When an event occurs in the kernel (e.g., a new connection is established on the TCP port dedicated to HTTP, or a packet is received on the UDP port for NFS), VINO spawns a worker thread and begins a transaction. It then invokes the grafted function (passing it a file descriptor or other data required to process the event).

```

// http server installation,
// invoked at user level
graftpoint_handle_o *gp;

gp = graft_namespace->lookup("tcp/80");
gp->add("http_server.o");

-----
// http server code, run as graft in kernel.
http_server(file_o *fd)
{
    char buf[256];
    fd->read(buf, sizeof(buf));
    // process http request...
    ...
}

```

Figure 2. Event grafting example. The first code fragment installs the server code on TCP port 80; the second code fragment represents the server itself. Each time a new connection is accepted on TCP port 80 a worker thread starts a new transaction and invokes the `http_server()` function, passing it the file descriptor from which `http_server()` can read the http request. When `http_server()` finishes handling the connection, it returns, and the worker thread commits the transaction and closes the connection.

When the grafted function returns, the worker thread commits the transaction and exits. An example of adding a function to an event graft point, and the outline of an HTTP server graft, are shown in Figure 2.

3.6 Summary

Returning to Table 1, we can now identify how VINO copes with the various classes of misbehavior. By design, our kernel is preemptible. Therefore, any thread, including any thread that called a graft, is preemptible (Rule 1). The combination of transaction abort and resource accounting protects against resource hoarding (Rule 2). If a graft consumes too many resources or runs for too long a period of time while holding a high-contention lock, its transaction is aborted. When a graft transaction is aborted, the graft is forcibly removed from the kernel, so that new invocations of the call use normal kernel code and not the misbehaving graft code. Our SFI compiler generates instructions to prevent grafts from accessing memory to which they are not entitled (Rule 3) and from executing functions to which the graft does not have access.

Rules 4 and 7 are provided for by a combination of static and dynamic methods. When constructing the list of graft callable functions, we must exclude those functions that return data without checking for appropriate permissions (Rule 4). MiSFIT and the dynamic linker ensure that only functions on the graft callable list are invoked by grafts (Rule 7).

Rules 5 and 6 are enforced statically through our downloading mechanism. In addition to verifying that a graft does not call inappropriate functions, the dynamic

loader prohibits grafting onto restricted kernel entry points, such as the security enforcement modules (Rule 5). The digital signature scheme described in Section 3.3 ensures that the kernel does not execute any grafts that are not known to be safe (Rule 6).

We believe that the combination of resource accounting, the downloading mechanism, and the separation of global and local policy decisions limits the applications affected by malicious grafts to only those applications that use those grafts and ensures that the kernel can make forward process, even in the presence of a malicious graft (Rules 8 and 9).

4 The Cost of Graft Protection

In previous work we presented a taxonomy of types of kernel extensions [16], and we use that taxonomy here to evaluate the overhead of graft maintenance in VINO. We identified three basic graft structures, each of which encompasses a broad class of kernel graft points. *Stream Grafts* act much like UNIX filters, accepting data, transforming or manipulating the data, and producing either a new data stream or result. Some examples of Stream Grafts are encryption, compression, and checksum calculation. A *Prioritization Graft* chooses a candidate from a set such as selecting a process to schedule, a page to evict, or a buffer to flush. A *Black Box Graft* is more general than Prioritization and Stream grafts; a Black Box graft has some number of inputs, some state, and a single output. From outside the graft, it appears as a “black box” function, producing a single output value. File system read-ahead, access control checking, and name resolution are examples of Black Box grafts.

In this section, we present sample grafts from each class and quantify the overhead associated with making the graft safe. The VINO system is still in its infancy, so we cannot run large, complex applications. For this reason, we perform the analysis at the graft level, as opposed to the application level. This allows us to perform fine grain measurements and also makes our measured overheads as conservative as possible. For example, if our protection mechanisms impose a 25% penalty on the graft in isolation, the observed penalty in a complete application can only be smaller.

Table 2 outlines our measurement methodology, identifying how we decompose each graft to isolate individual overhead components. Figure 3 depicts the code paths and general structure of our grafts, highlighting the typical paths that we measure. In an effort to encapsulate the full cost of extensibility, we measure our *base path* by removing any levels of indirection and results checking that we introduced to facilitate grafting. The *VINO path* measures our normal kernel paths; it includes any extra levels of indirection

we impose, but no transaction overhead. The *null path* includes full support for grafting, including transaction begin and end, but does the minimal amount of work possible for each example graft. The *safe* and *unsafe paths* include the full graft path, with transactions, and quantify the MiSFIT overhead in the difference between the two paths. Finally, the *abort path* results from a transaction abort at the end of the graft execution in the *safe path*. As we measure the cost of these increasingly complex execution paths, we report both the total execution time of each path and the incremental overhead between successively complex paths.

Measurement	Explanation
Base path	Kernel code path with all extra indirection and graft-support removed.
VINO path	Normal VINO kernel path, with indirection for graft support and return-value verification.
Null path	Includes graft stubs, transaction begin and commit, and minimal (null) graft.
Unsafe path	Includes full graft code and lock overhead
Safe path	Includes code protected with MiSFIT.
Abort path	Complete safe path with transaction abort instead of commit.

Table 2. Measurement Methodology. Each graft benchmark will decompose the graft cost into the components described here.

Our test platform consists of an Intel Endeavor motherboard with a 120 MHz Pentium processor, a 512 KB pipeline burst L2 Cache, and 32 MB of 60ns EDO DRAM. We use a single 5400 RPM Fujitsu M2694ESA disk with a SCSI interface, a formatted capacity of 1080MB, an average seek time of 9.5 ms, and a 64KB buffer. As our tests were performed on a Pentium, we were able to take advantage of the hardware cycle counter on the CPU. We computed the number of cycles for each test, and then using the clock speed of the processor, converted from cycles to microseconds. To reduce the sensitivity of our results to cache effects, we drop outliers by eliminating the top 10% and bottom 10% of the measurements before computing the means and standard deviations. (We ran each test between 300 and 3000 times depending on the test.) In most cases the standard deviations were negligible (less than 2.5% of the mean). We observed higher standard deviations for very short duration events, because an individual cache miss can account for a significant fraction of the measurement. In a few of the tests, we still find differences in cache behavior between test cases; in

these cases, we explicitly measure and report the additional cache overheads.

For each of our sample grafts, we perform a simple cost-benefit analysis. In each case, the cost of the graft is the time to execute the grafted function, along with the general overhead of executing a graft (transaction protection, MiSFIT overhead, etc.). The difference between the *safe path* and the *VINO path* provides the total cost of a graft in terms of the overhead that it adds to the system. The difference between each successive pair of measurements in Table 2 corresponds to one part of the overhead. The *null path* adds the cost of transaction protecting a graft function to the *VINO path*. The cost of the graft function itself can be determined by comparing the *null path* with the *unsafe path* (where we add the graft function without any MiSFIT protection). Finally, we compute the MiSFIT overhead by comparing the *unsafe path* and the *safe path* measurements.

The benefits associated with each graft depend on the specific functionality being grafted into the kernel. We estimate the benefit derived from each of our test grafts and compare this to the cost of the graft in order

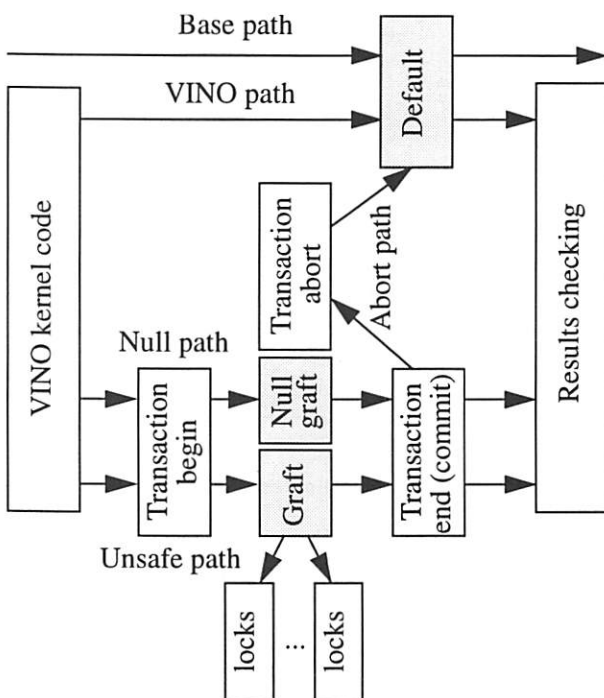


Figure 3. Graft Evaluation Model. The *base path* represents the cost of kernel functionality without the extra indirection required to support grafting. The *VINO path* includes the cost of indirection. The *null path* includes the graft overhead and any additional overhead required to check the graft's return value(s), but not the actual graft code. The *unsafe path* includes the actual graft, but does not include MiSFIT overhead, and the *abort path* is the complete, *unsafe path* with transaction commit replaced by a transaction abort. The shaded boxes represent points where policy decisions are made.

to determine the suitability of the VINO extension architecture for the different types of grafts that we have measured.

4.1 A Black Box Graft: Read-Ahead

File read-ahead is an example of a policy for which traditional operating systems implement a general algorithm that is good for most applications, but not necessarily optimal for all. A typical file read-ahead policy operates on the assumption that most applications perform sequential I/O. When the system detects sequential access to a file, it asynchronously prefetches some additional amount of file data with each read request. Because most file accesses are sequential [3], this policy usually improves performance. There are several cases, however, where this general policy does not improve (and can even degrade) application performance [12].

At first glance, it may seem that user-level threads are the simple and obvious solution to application-directed prefetching. However, without kernel support, a strictly user-level prefetching implementation is unable to exploit kernel-level information about the on-disk layout of the file data. Similarly, without explicit kernel support, an application that performs many short sequential reads to different offsets in a large file may incur the overhead of having the kernel prefetch unneeded parts of the file. Finally, thread-level prefetching is indistinguishable from normal user I/O and the kernel is unable to assign accurate priorities to pending I/O requests.

4.1.1 Cost-benefit Analysis of Read-ahead

File read-ahead is one of the most appealing kernel policies to graft, because the potential gains are large. For the remainder of this discussion, we will consider read-ahead in the context of a specific hypothetical application, modeled on a random access workload such as might be generated by a database server. The core of this application is a loop that reads a block of data, and then performs some computation on it. We assume that the application reads the blocks of data in a non-sequential order, but has advance knowledge of what blocks it will need.

Because the default kernel read-ahead policy only supports sequential access, no prefetching would normally be done. Thus, each time the application issued a non-sequential read request, it would block until the data was fetched from disk. If the application can graft a new read-ahead function onto the file, however, then each time it reads one block, it can also prefetch the next block. Thus, we can imagine the application reading block A and prefetching block B,

computing on block A, then reading block B and prefetching block C, and computing on block B.

What are the costs and benefits associated with this scheme? Consider the request to read block B. The cost for the application is the overhead associated with prefetching the block as described above. The benefit to the application is that the amount of time that the application is suspended waiting for block B is reduced by the amount of time since that prefetch request. This is the amount of time that the application spent computing between read requests. Thus, the application will win if the cost of the read-ahead graft is less than the time the application spends between read requests.

4.1.2 Implementing the Read-ahead Graft

In VINO, application level file descriptors are handles for kernel level *open-file* objects. Traditional file-related system calls (read, seek, etc.) are translated to method invocations on the appropriate open-file. Whenever a user issues a read request, the corresponding method on the open-file handles the read, and then calls its *compute-ra* method to determine which (if any) additional file blocks should be prefetched. This function is passed a descriptor describing the offset and size of the current read request, and is allowed to provide a list of additional file extents that should be prefetched. These prefetch requests are passed to the underlying file system where they are added to a per-file prefetch queue. The file system removes prefetch requests from this queue and issues them to the I/O system as memory becomes available for read-ahead. (In this manner, if a graft of the *compute-ra* function asks for 100MB to be prefetched, it will not steal all of the system's memory pages. Instead, the 100MB will be prefetched in order, as pages become available.) The allocation of memory buffers to satisfy read-ahead requests is determined by a global policy that cannot be grafted by users with normal privileges.

The default read-ahead policy used by VINO only prefetches when the user accesses a file sequentially. Applications that wish to specify an alternate prefetching policy do so by grafting a new *compute-ra* function onto the appropriate open-file object.

As described above, our hypothetical application would benefit from a read-ahead policy that permitted it to specify the blocks to be prefetched. To this end, we implemented a graftable read-ahead policy for non-sequential access. A memory buffer is shared between the application and the read-ahead graft, allowing the application to specify its anticipated file access pattern. The graft version of the *compute-ra* function uses the data in this shared buffer to issue read-ahead requests.

4.1.3 Measuring Read-ahead Graft Overhead

We tested the read-ahead graft by reading three thousand four kilobyte blocks in a random order from a twelve megabyte file. Each time the application code issued a read request to the open file object, it also placed the location and size of its subsequent read in the shared buffer so that it could be prefetched. Table 3 shows the overhead for the read-ahead graft.

	Overhead (μ s)	Elapsed time (μ s)
Base path		0.5
<i>Indirection cost</i>	1.0	
VINO path		1.5
<i>Transaction begin</i>	36.0	
<i>Null graft cost</i>	1.5	
<i>Transaction commit</i>	+ 28.0	
<i>Incremental overhead</i>	65.5	
Null path		67.0
<i>Lock overhead</i>	33.0	
<i>Graft function</i>	2.0	
<i>L1 cache miss time</i>	+ 2.0	
<i>Incremental overhead</i>	37.0	
Unsafe path		104.0
<i>MiSFIT overhead</i>	3.0	
Safe path		107.0
<i>Abort cost (additional, above commit time)</i>	1.0	
Abort path		108.0

Table 3. Read-ahead Graft Overhead. The read-ahead graft decides which page should be prefetched with each read request. The *base path* measures the time to select the next (i.e., sequential) block. The graft function allows the user to specify an access pattern. This simple function has a short execution time, yielding a disproportionately large graft overhead.

From these measurements, we see that the cost of executing the grafted read-ahead function (the difference between the *unsafe path* and the *null path*) is 37 μ s. Most of this cost is the overhead of acquiring a lock before accessing the shared memory buffer. The other cost of executing the grafted read-ahead function is the grafting overhead—the cost of running the graft function in a transaction, the overhead of applying MiSFIT to the graft function, and the extra checking required to validate the values returned by the graft function. Table 3 shows that the total cost of starting and committing a transaction is 64 μ s. The overhead

imposed by MiSFIT, derived by comparing the measurements of the safe path and the *unsafe path*, is another three microseconds. Thus the total grafting overhead for this function is 65.5 μ s.

Returning to our cost benefit analysis, the total cost of executing the grafted read-ahead function (the *safe path* from Table 3) is 107 μ s. Thus, our application would benefit from using this graft assuming that it spends at least 107 μ s between read requests. For comparison, it takes 137 μ s to sum a four kilobyte array of integers on our test machine. (4KB is our file system block size.)

4.2 A Prioritization Graft: Page Eviction

Virtual memory page eviction is another example of a policy for which traditional operating systems implement a general algorithm (e.g., some variant of the clock algorithm) that is good for most applications, but not all. Applications for which LRU is the right paging strategy will enjoy fine performance under this algorithm, but there are cited cases where such an algorithm is suboptimal [2,12,18].

The key challenge in supporting application-provided page out selection is to do so in a manner that does not compromise the integrity of the virtual memory system. There are three requirements necessary to enforce this. First, the page eviction decision must be made in a timely fashion, because poor VM performance can slow the entire system. Second, the value returned by the graft must be valid (or detectably invalid). Third, the graft cannot permit the application to use more physical memory than would be allowed if the same application ran without a page eviction graft.

4.2.1 VINO VM Page Eviction

The VINO virtual memory system is based loosely on the Mach VM system [14]. A virtual address space (VAS) consists of a collection of memory objects mapped to virtual address ranges. A memory object represents a contiguous piece of data that may be backed by a variety of objects such as a device, a network connection, or a file. Once a memory object is associated with a particular object, the object becomes responsible for handling page faults to the memory object in a manner appropriate for the materialized item (e.g., read a file from disk, read data from a network connection).

Virtual memory page eviction is implemented by a two-level eviction algorithm. A global page eviction algorithm selects a victim page. Then, if the owning VAS has installed a page eviction graft, it invokes the graft passing it the victim page and a list of all other pages that the virtual memory system currently assigns to the particular VAS. The VAS-specific function can

accept the victim page or suggest another page as a replacement (similar to Cao's replacement strategy [5]).

The global algorithm then verifies that the selected page belongs to the specific VAS and is not wired. If either of these checks fails the system ignores the request and evicts the original victim. When an acceptable choice is returned, we use Cao's approach and place the original victim into the global LRU queue in the spot occupied by the replacement specified by the graft.

	Overhead (μ s)	Elapsed time (μ s)
Base path		39
<i>Indirection cost</i>	1	
VINO path		40
<i>Transaction begin</i>	52	
<i>Null graft cost</i>	2	
<i>Transaction commit</i>	34	
<i>Results checking</i>	+2	
<i>Incremental overhead</i>	90	
Null path		130
<i>Lock overhead</i>	34	
<i>Graft function</i>	160	
<i>Results checking</i>	+5	
<i>Incremental overhead</i>	199	
Unsafe path		329
<i>MiSFIT overhead</i>	26	
Safe path		355
<i>Abort cost (additional, above commit time)</i>	-7	
Abort path		348

Table 4. Page Eviction Graft Overhead. The grafted code runs in addition to the default code. Because the pagedaemon runs as a background thread, its behavior is not completely deterministic, and we observe high standard deviations when calculating incremental overheads. While the measurements with over 329 μ s duration had less than 3% standard deviation, short-duration measurements had high standard deviations (the highest was 16% for the *VINO path*). For both *unsafe* and *safe paths*, the graft overrules the default victim selection. The total cost of the *abort path* is lower than the *safe path*, because results checking and list manipulation are simplified.

4.2.2 Measuring Page Eviction Graft Overhead

We tested our sample page eviction graft with an application that has a 2MB data footprint of which a few pages are performance critical. The application and graft share a region of memory in which the application places the page numbers of those pages it wishes to

retain in memory. During page out, the graft checks the globally selected victim to ensure that it is not one of the pages listed by the application. If it is, the graft scans the list of pages that it is allowed to evict, returning the first page it finds that is not on its list of important pages. Table 4 shows the different measurement paths for this graft.

When a graft disagrees with the default victim selection, the cost of victim selection increases by an order of magnitude, but is quite reasonable compared to the cost of the I/O operation that might be saved. If we apply a cost-benefit analysis, the cost of adding the graft is 316 μ s, while the benefit of avoiding a page fault is approximately 18 ms in our system. The graft can disagree with the victim selection approximately 57 times for each I/O that we save. In addition, the cost is reduced to 159 μ s when the graft agrees with the default victim selection, and because the pageout daemon runs asynchronously, the increased cost of victim page selection is unlikely to reduce application performance.

4.3 A Prioritization Graft: Scheduling

It is often the case that a group of threads or processes work in concert, and should be scheduled as a group. For example, a database server process and its clients can be thought of as a single application; when there are no outstanding server requests, the server process should not be scheduled, but when several clients are blocked on requests to the server, the server process should be given a proportionally larger share of the total CPU in order to more quickly reply to the outstanding requests.

Each user-level process has associated with it a kernel-level thread. When the kernel thread is chosen to be run next, its *schedule-delegate* function is run. The default version of this function returns the identity of the thread itself (i.e., instructions to run the selected thread). The *schedule-delegate* function can be replaced by grafting a process-specific function that, in the example above, would have the client return the identity of the server process when the client was waiting for the server to reply to a request.

Our example *schedule-delegate* graft scans a process list of 64 entries, examines each (to determine if one of the other processes should be run instead) and then returns its own ID.

The *base path* measurement is the cost of switching processes on our system, the primary costs of which are choosing which thread to run next, switching kernel threads, and switching VM contexts. In this case, the *VINO path* differs from the *base path* only in a call to a function that returns the new threads's ID and the code to verify that the returned ID is that of a valid thread (which is accomplished by probing a hash table

containing the valid thread IDs). The *null path* adds transaction support around an invocation to this trivial function. The *unsafe path* invokes the graft described above without SFI protection, and the *safe path* includes the cost of SFI protection. The results are presented in Table 6.

	Overhead (μ s)	Elapsed Time (μ s)
Base path (two switches)		54
Indirection cost	1	
VINO path		55
Transaction begin	38	
Null graft cost	2	
Transaction commit	30	
L1 cache miss time	+6	
Incremental Overhead	76	
Null path		131
Lock overhead	33	
Graft function	35	
Result checking	+4	
Incremental Overhead	72	
Unsafe path		203
MiSFIT overhead	5	
Safe path		208
Abort cost (additional, above commit time)	3	
Abort path		211

Table 5. Scheduling Graft Overhead. The *base path* measurement includes the time to select the next process to run, switch to it, and switch back (including switching VM contexts twice). The *null path* includes a call to a function that returns its argument (the candidate thread). The *unsafe path* adds the invocation of a non-trivial function that locks and searches the process list. The largest increase in overhead comes from the transaction and lock costs, which sum to twice the process switch cost.

Because this graft walks the process list, it must acquire a lock for the list. The *unsafe* and *safe paths* add this cost to the cost of the code that walks the process list; the *safe path* adds the cost of MiSFIT protection. Each iteration of the loop that walks the 64-element process list takes about 0.5 μ s, primarily because our collection class implementation is not well-optimized.

The cost for this graft, starting with the fixed transaction begin/commit cost, is higher than the *base path* cost (for switching processes twice). Although twice the cost of a process switch, it is still roughly 2% of a typical timeslice of 10 ms (as opposed to 0.5% for the *base path*).

The benefit of permitting processes to control scheduling is difficult to quantify; however, the benefit of being able to control which process runs next can be considerable. Multimedia applications are often structured as several cooperating processes or threads. In a conventional system, if the user interface thread is scheduled when it comes time for the application to display the next video frame, the best the UI thread can do is yield, and hope that the video thread is scheduled soon. With the ability to delegate a timeslice in the manner discussed here, the UI thread could hand off directly to the video thread, with the goal of better meeting the scheduling deadlines of the application.

An operating system with support for real-time scheduling and service guarantees might better meet the needs of this particular application; however, we do not believe that we can *a priori* determine all desirable scheduling policies and hard-code them into the kernel.

4.4 A Stream Graft: Encryption/Decryption

A stream graft is used to transform a data stream as it passes through the kernel. Examples of stream grafts are compression (and decompression), logging, mirroring, and encryption (and decryption).

Our graft performs a trivial (xor-style) encryption of data as it is copied to user level, and symmetrical decryption as it is brought into the kernel from user level. The encryption algorithm used is not computationally intensive, which is a conservative position to take. The primary cost imposed by our software fault isolation tool is protection against errant loads and stores, so the higher the ratio of memory accesses to other instructions, the higher the SFI overhead. The most trivial stream graft just copies data from input to output without transforming it; this graft has the highest ratio of stores to other instructions. Therefore, the simpler the transformation the graft performs, the more conservative the overhead estimate. For example, the cost of a computationally intensive encryption scheme (e.g., DES) would dwarf the overhead associated with software fault isolation.

Our sample graft is passed an 8KB input data buffer block and an 8KB output buffer. The graft encrypts the data into the output buffer and returns. This graft is particularly interesting in that it requires no synchronization overhead (the input and output buffers have been obtained in the caller), but offers nearly the worst case of software fault isolation overhead, because it consists almost entirely of load and store instructions. Table 6 shows the overhead for the encryption graft.

For the *base path* measurement we use the in-kernel bcopy function to copy an 8KB buffer (105 μ s). The *VINO path* adds a function indirection that is sufficiently fast to be undetectable, and, as above, the

null path adds transaction begin and commit (64 μ s). The *base* and *VINO path* measurements are artificially low because they call bcopy in a tight loop. Using the Pentium on-chip counters, we measured an additional 24 μ s spent servicing L1 cache misses in the *null path* case, for a bcopy time of 193 μ s.

	Overhead (μ s)	Elapsed time (μ s)
Base path		105
VINO path		105
Transaction begin	32	
Transaction commit	32	
L1 cache miss time	+24	
Incremental overhead	88	
Null path		193
Graft function	166	
Unsafe path		359
MiSFIT overhead	187	
Safe path		546
Abort cost (additional, above commit time)	4	
Abort path		550

Table 6. Encryption Graft Overhead. As expected, this graft is a worst-case scenario for software fault isolation, imposing more than 100% overhead on the graft function.

The *unsafe path* encrypts the data as it copies it from input to output, adding another 166 μ s over and above the cost of the bcopy, for a total of 359 μ s. The encryption takes 3.4 times that of a straight bcopy (which is implemented using a hardware copy instruction that has a cost of only one cycle per word copied).

The cost of MiSFIT protection on the *safe path* adds 187 μ s, for a total of 546 μ s, or 5.2 times a straight bcopy. This overhead is not surprising, given the lack of optimization in our software fault isolation tool. Our tool protects each indirect memory access; since the graft consists primarily of memory accesses, we see a protection overhead between two and three times the cost of the function itself.

4.5 Transaction Failure Overhead

In each of our sample grafts, we measured the time required to abort the graft. This cost is a function of the number and complexity of the undo functions, the number of locks to release, and the constant overhead associated with ending a transaction. This cost varies dramatically, depending on the complexity of the graft.

For each of the grafts described above, we measured the cost of aborting the *null path* as well as the full grafted path. These measurements are shown in Table 7.

Our sample grafts have sufficiently little state that the full abort cost is only 0% to 40% more than the null abort cost. Most of these grafts have little *undo* work and few locks. While we believe that these grafts are representative of the fine-grain grafts that VINO allows, more complex grafts will have higher abort costs. The total abort time is represented by the equation: $\text{abort overhead} + \text{unlock cost} + \text{undo cost}$. The abort overheads we measured ranged from 32–38 μ s, and we measured the cost of releasing a lock at 10 μ s per lock. The undo cost should be somewhat less than the actual cost of running the graft. Therefore, the abort cost equation becomes: $35\mu s + 10L + cG$, where L is the number of locks to be released, G is the cost of the graft, and c is a constant less than one.

	Null Abort (μ s)	Full Abort (μ s)
Read-Ahead	32	45
Page Eviction	38	50
Scheduling	33	45
Encryption	36	36

Table 7. Graft Abort Costs. For each of our sample grafts, the difference between the two columns is a function of the number and complexity of the undo functions and the number of locks that must be released.

The most significant variable in aborting a transaction occurs when the graft hoards resources and must be timed out. We currently schedule time-outs on system-clock boundaries, which occur every 10 ms. Therefore, the delay for timing out a transaction will be between 10 and 20 ms. This is obviously too coarse grain for some resources, and we expect to experimentally determine a more appropriate timing as the system matures.

4.6 Summary

The overhead associated with using transactions and software fault isolation to protect kernel integrity from misbehaving grafts varies according to the type of actions performed by the graft. As the encryption graft demonstrated, MiSFIT can increase the execution time of graft code by nearly 200%. For less data intensive grafts, such as the file read-ahead graft, the MiSFIT overhead, while large relative to the cost of the graft itself, is only a few microseconds. Transaction costs are relatively stable across all grafts, increasing in proportion to the number of locks acquired on a graft's behalf. Each use of a

transaction lock instead of a conventional kernel mutex lock adds approximately 19 μ s to the graft's execution time and 14 μ s to the abort cost.

The measured cost of running a graft in the context of a transaction can be substantial, adding as much as 200 μ s to the execution time of the graft code. The true cost of downloading user code into the kernel, however, must be measured in terms of the performance and functionality gained by allowing applications to modify the kernel. Grafts may eliminate or hide disk accesses, avoid context switches, or eliminate programmer labor by allowing the reuse of kernel functionality. In many cases, the time gained more than compensates for the overhead of the grafting mechanisms. In other cases, the gains in flexibility and savings in labor will be sufficient compensation, and in some cases, the cost will outweigh any potential benefits, and we must explore other alternatives for kernel extensibility.

5 Related Work

VINO is one of many new operating systems that belongs to the class of extensible systems. It is most similar to the SPIN system [4]. In SPIN, extensions are written in a typesafe language (Modula-3) and downloaded into the kernel where they initiate a thread. Once installed, the thread can install handlers for any kernel events for which it has appropriate permission and in which it is interested. The use of a typesafe language simplifies some of the safety issues involved in building an extensible system, because the extensions cannot reference disallowed interfaces or data. Cleaning up after errant extensions is also simplified in SPIN, because the Modula-3 garbage collector can clean up state when a graft terminates. However, the areas we found most challenging, such as detecting and dealing with resource hoarding, identifying malicious extensions, and identifying the set of graft-callable and graft-replaceable interfaces, are also challenges for SPIN.

Our event graft model is similar to the event model of SPIN. Where the function graft model (discussed in section 3.4) is appropriate for simple, fine-grained graft points that correspond to single functions, event graft points provide better support for the addition of new services to the kernel.

The Exokernel project [6] is an extreme example of an extensible system. The goal of the Exokernel project is to remove abstractions from the kernel and export a low-level machine interface directly to applications. User-level libraries implement most of the abstractions traditionally implemented by the kernel, and the kernel implements the bare-minimum functionality required to export the hardware interface

to applications safely. There are two ways to extend the Exokernel. The first is to modify the user-level libraries that implement the kernel abstractions. Because VINO is a conventional kernel architecture, there is no analogy in VINO. The second method of extending Exokernel is to download code into the kernel and use software fault isolation, as is done in VINO, to ensure safety.

Another approach to extensibility is to provide an interpretive environment in the kernel in which kernel extensions can be run. The interpreter can ensure safety by preventing extensions from wreaking havoc in the main kernel, but often incurs a significant runtime overhead [16].

The adaptable systems, such as Synthetix [19], take a different approach from the extensible systems. Rather than having applications explicitly modify the kernel's behavior, Synthetix is designed so that commonly executed paths through the operating system can be specialized. For example, the common path through the file system accesses the same file descriptor and the same or sequentially next block in the designated file. By providing a specialized component that removes branches and the normal code to map file descriptors to kernel structures, the performance of the normal case can be greatly improved [13]. The only additional cost comes in the form of checks that distinguish between the normal path and the specialized path and allow the system to execute the correct one at the correct time. This approach is sufficient for improving performance for paths that already exist in the kernel, but does not address functionality that is not present in the kernel initially.

6 Lessons Learned

C++ bought us some headaches that a safe language such as Modula-3 would have avoided. In particular, garbage collection would have simplified our task of cleaning up after a misbehaved graft aborts. We could also have avoided work-arounds such as delaying deletes until transaction abort. Finally, the SFI overhead for data intensive grafts, such as encryption, is irritating; a more constrained language would have provided protection at the compiler level. While we have succeeded in making our kernel robust against malicious grafts, it has been painful at times.

By far, the most challenging issues were not language issues; they were system design issues. Repeatedly, we found ourselves making trade-offs between restricting the graft interface and adding overhead to handle all the types of malice we foresaw with the broadened interface we provided. In general, we strove to make grafts as flexible as possible, even when it bought us extra complexity in the design or

```
get_lock(requested_lock) {
    for (lock=head(holders_list);
        lock != NULL;
        lock=lock->next)
        if (is_conflict(lock,
                        requested_lock)) {
            append(lock, waiters_list);
            break;
        }
    if (lock != NULL)
        append(lock, holders_list);
}
```

Figure 4. Conventional lock algorithm.

extra overhead in the protection mechanisms. The system is still too young for us to determine whether these trade-offs were justified. As we gain more experience with sophisticated applications, we can reevaluate this design decision.

We also discovered that we had to think very differently when designing a system for fine grain extensibility. Every decision that might conceivably be extended had to be encapsulated in an interface. This encourages extreme modularity. (At this point, we have not been able to completely quantify this cost.) For example, a conventional lock manager might implement the `get_lock` request as shown in Figure 4. Unfortunately, this code encapsulates at least two policy decisions. First, it assumes that any incoming lock request can be granted if it does not conflict with any holders, ignoring the locks on the wait list (e.g., it implements a reader priority locking protocol). Second, it assumes that locks should be appended to the waiters list, implying an ordering. A more general implementation of `get_lock` is shown in Figure 5. This implementation encapsulates each policy decision at the cost of a level of indirection at each decision point. On our system, function calls typically cost approximately 35 cycles at 8.3 ns/cycle; these add up remarkably quickly.

Perhaps the most daunting design issue that confronted us was selecting the right abstraction for grafts. Are they threads? Are they simple functions? We revisit this question regularly, but our current position is that grafts are effectively user-level processes that happen to run in the kernel's address space. As processes are isolated from the kernel by address space boundaries, grafts are isolated from the kernel by software fault isolation. Grafts interact with the kernel

```
get_lock(requested_lock) {
    if (can_grant = grantable(lock))
        lock_add(lock, holders_list);
    else
        lock_add(lock, waiters_list);
}
```

Figure 5. Encapsulated lock algorithm.

through a selected set of interfaces, but these interfaces are much lower level and functionally richer than the processes' system call interface. In an ideal world, grafts should look just like other kernel code, and for the most part, they do.

7 Conclusion

Two simple mechanisms, software fault isolation and transactions, protect our kernel from mischievous extensions imposing penalties ranging from 104 to 270 μ s. In all of our test cases, these costs are outweighed by the potential benefits of the grafts. Because grafts either provide functionality not present in the system or significantly improve performance, we believe that such overhead is acceptable for most scenarios. It is certainly possible that we have overlooked classes of misbehavior that we cannot detect and/or handle, but our mechanisms are applicable across a wide range of extensions.

Acknowledgments

We would like to thank the program committee and reviewers for their helpful comments, and especially our shepherd, Jim Gray, for his insight and assistance in improving the paper and its presentation. We would also like to thank the members of the VINO group, especially Aaron Brown and David Holland, for their help.

References

- [1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, Atlanta, GA, July 1986, 93–112.
- [2] Appel, A., Li, K., "Virtual Memory Primitives for User Programs," *Proc. ASPLOS IV*, Santa Clara, CA, April 1991, 96–107.
- [3] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proc. 13th SOSP*, Pacific Grove, CA, Oct. 1991, 198–212.
- [4] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP*, Copper Mountain, CO, Dec. 1995, 267–284.
- [5] Cao, P., Felten, E., and Li, K., "Application-Controlled File Caching Policies," *Proc. 1994 Summer USENIX Conf.*, Boston, MA, June 1994, 171–182.
- [6] Engler, D., Kaashoek, F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th SOSP*, Copper Mountain, CO, Dec. 1995, 251–266.
- [7] Haskin, R., Malachi, Y., Sawdon, W., and Chan, G., "Recovery Management in QuickSilver," *ACM TOCS* 6, 1, Feb. 1988, 82–108.
- [8] Illustra Information Technologies, "Introduction to Illustra," Part No. ILL0795-01III, *Illustra Web DataBlade User's Guide*, Release 2.1 Beta. Sep. 1995. Part No. WEB-00-12-UG.
- [9] Montz, A., Mosberger, D., O'Malley, S., Peterson, L., Proebsting, T., Hartman, J., "Scout: A Communications-Oriented Operating System," Department of Computer Science, University of Arizona, Technical Report 94-20, June 1994.
- [10] Microsoft Corp., "How Software Publishers Can Use Authenticode Technology," <http://www.microsoft.com/intdev/signcode>.
- [11] Nelson, G., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [12] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J., "Informed Prefetching and Caching," *Proc. 15th SOSP*, Copper Mountain, CO, Dec. 1995, 79–91.
- [13] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th SOSP*, Copper Mountain, CO, Dec. 1995, 314–324.
- [14] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Proc. ASPLOS II*, Palo Alto CA, Oct. 1987, 31–39.
- [15] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the Architecture of the VINO Kernel," Harvard University Computer Science Technical Report 34-94, 1994.
- [16] Small, C., Seltzer, M., "A Comparison of OS Extension Technologies," *Proc. 1996 USENIX Conf.*, San Diego, CA, Jan. 1996, 41–54.
- [17] Small, C., "MiSFIT: A Minimal i386 Software Fault Isolation Tool," Harvard University Computer Science Technical Report TR-07-96, 1996.
- [18] Stonebraker, M., "Operating Support for Database Management," *CACM* 24, 7, July 1981, 412–418.
- [19] Volanschi, E., Muller, G., Consel, C., "Safe Operating System Specialization: the RPC Case Study," *Proc. 1st Workshop on Compiler Support for System Software*, Tuscon, AZ, Feb. 1996.
- [20] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proc. 14th SOSP*, Asheville, NC, Dec. 1993, 175–188.
- [21] Waldspurger, C., Weihl, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. 1st OSDI*, Monterey, CA, Nov. 1994, 1–11.

Safe Kernel Extensions Without Run-Time Checking

George C. Necula Peter Lee

*School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891
{necula,petel}@cs.cmu.edu*

Abstract

This paper describes a mechanism by which an operating system kernel can determine with certainty that it is safe to execute a binary supplied by an untrusted source. The kernel first defines a safety policy and makes it public. Then, using this policy, an application can provide binaries in a special form called *proof-carrying code*, or simply PCC. Each PCC binary contains, in addition to the native code, a formal proof that the code obeys the safety policy. The kernel can easily validate the proof without using cryptography and without consulting any external trusted entities. If the validation succeeds, the code is guaranteed to respect the safety policy without relying on run-time checks.

The main practical difficulty of PCC is in generating the safety proofs. In order to gain some preliminary experience with this, we have written several network packet filters in hand-tuned DEC Alpha assembly language, and then generated PCC binaries for them using a special prototype assembler. The PCC binaries can be executed with no run-time overhead, beyond a one-time cost of 1 to 3 milliseconds for validating the enclosed proofs. The net result is that our packet filters are formally guaranteed to be safe and are faster than packet filters created using Berkeley Packet Filters, Software Fault Isolation, or safe languages such as Modula-3.

1 Introduction

In this paper we address the problem of how an operating-system kernel or a server can determine with

absolute certainty that it is safe to execute code supplied by an application or other untrusted source. We propose a mechanism that allows a kernel or server—from now on referred to as the *code consumer*—to define a safety policy and then verify that the policy is respected by native-code binaries supplied to it by an untrusted *code producer*.

In contrast to some previous approaches, we do not rely on the usual authentication or code-editing mechanisms. Instead, we require that the code producer creates its binaries in a special form, which we call *proof-carrying code*, or simply PCC. A PCC binary contains an encoding of a formal proof that the enclosed native code respects the safety policy. The proof is structured in such a way that makes it easy and foolproof for any agent (and in particular, the code consumer) to verify its validity *without* using cryptographic techniques or consulting with external trusted entities; there is also no need for any program analysis, code editing, compilation, or interpretation. Besides being safe, PCC binaries are also extremely fast because the safety check needs to be conducted only once, after which the consumer knows it can safely execute the binary without any further run-time checking.

In a PCC binary, the proof is linked with the native code so that its validity guarantees the code's safety. Furthermore, proof-carrying code is tamper-proof; the consumer can easily detect most attempts by any malicious agent to forge a proof or modify the code. Tampering can go undetected only if the adulterated code is *still* guaranteed to respect the consumer-defined safety policy. Another feature of the PCC method is that the proof checking algorithm is very simple, allowing fast and easy-to-trust implementations.

The safety policy is defined and published by the code consumer and comprises a set of proof-formation rules, along with a set of preconditions. Safety policies can be defined to stipulate standard

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

requirements such as memory safety, as well as more abstract and fine-grained guarantees about the integrity of data-abstraction boundaries. To take a simple example, consider the abstract type of file descriptors. In this case, a client is said to preserve the abstraction boundaries if it does not exploit the fact that file descriptors are represented as integers (by incrementing a file descriptor, for example).

Although we have worked out many of the theoretical underpinnings for PCC (and indeed, most of the theory is based on old and well-known principles from logic, type theory [4, 11], and formal verification [5, 6, 8]), there are many difficult problems that remain to be solved. In particular we do not know at this point the most practical way to generate the proofs. We have thus set out to gain some preliminary experience, both to measure the benefits and to identify the practical problems.

In the experiments reported in this paper, we have in fact achieved fully automatic proof generation. In general, however, this problem is similar to program verification and is not completely automatable. Actually, the problem is somewhat easier than verification because we have the option of inserting extra run-time checks (as is done in Software Fault Isolation), which would have the effect of simplifying the proving process at the cost of reducing performance. By “extra”, we mean run-time checks that are not intrinsically a part of the algorithm of the extension code. (For example, SFI will actually edit the code and insert “extra” checks; PCC does not normally do this.) Fortunately, we have not yet had any need or desire to insert extra run-time checks in any of our PCC examples. Still, automation of proof generation remains as one of the most serious obstacles to widespread practical application of PCC.

In our main experiment, we implemented several network packet filters [12, 15] in DEC Alpha assembly language [19] and then used a special prototype assembler to create PCC binaries for them. We were motivated to use an unsafe assembly language in order to place equal emphasis on both performance and safety, as well as to demonstrate the generality of the PCC approach. In addition to the assembler, we implemented a proof validator that accepts a PCC binary, checks its safety proof, and if it is found to be valid, loads the enclosed native code and sets it up for execution.

The results of this and other experiments are encouraging. For our collection of packet filters, we are able to automate completely the generation of the PCC binaries. The one-time cost of loading and checking the validity of the safety proofs is between

1 and 3 milliseconds. Because a safety proof guarantees safety, our hand-tuned packet filters can be executed safely in the kernel address space without adding any run-time checks. Predictably, they are much faster than safe packet filters produced by any other means with which we are familiar.

We believe that our early results show that proof-carrying code is a new point in the design space that is worthy of further attention and study. This paper presents an overview of the approach. We begin with a brief overview of the process of generating and validating the safety proofs. Then, we make this more concrete by showing how a safety policy can be defined and proofs created for a generic assembly language. This is followed by a description of our main experiment involving safe network packet filters. The benchmark results provide some preliminary indication that the PCC methodology has the potential to surpass traditional approaches from a safety point of view while maintaining or improving performance. In particular, we show that PCC leads to faster and safer packet filters than previous approaches to code safety in systems software, including Berkeley Packet Filters [12], Software Fault Isolation [23], and programming in the safe subset of Modula-3 [1, 9, 17]. Finally, we conclude with a discussion of the remaining difficulties and speculate on what might be necessary to make the approach work on a practical scale.

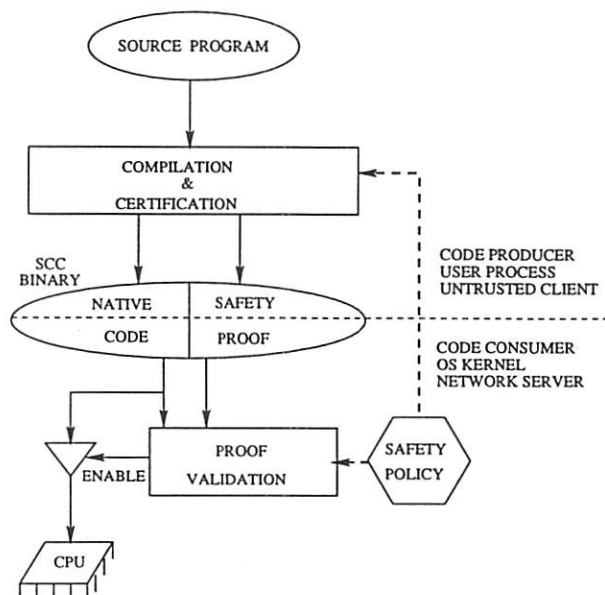


Figure 1: Overview of Proof-Carrying Code.

2 Proof-Carrying Code

Figure 1 depicts the process of generating and using a PCC binary. The process begins with the code consumer defining and publicizing a safety policy. This policy defines formally what is meant by “safety” and also specifies the interface between the consumer and any binary provided by the producer. Taking the policy into account, the code producer compiles (or assembles) and proves the safety of a source program, through a process which we call certification. This results in a PCC binary that can be delivered to the code consumer. Upon receipt, the consumer validates the safety proof enclosed in the PCC binary. Finally, if the proof is found to be valid, the code consumer can safely execute the native-code part of the PCC binary.

The following subsections describe each of these phases in more detail. The whole process is based on concepts from logic, semantics, and type theory, and so the rest of this section is necessarily somewhat technical, with most details beyond the scope of this paper. We will thus attempt to explain only the basic technicalities and key intuitions here. Those readers who would like more details on the underlying theory can find them in a separate technical report [16]. The impatient reader may want to skip ahead to Section 3 where we show, for the case of network packet filters, that proof-carrying code surpasses previous approaches in both safety and performance.

2.1 Defining a Safety Policy

The first order of business is to define precisely what constitutes safe code behavior. We do this by specifying a *safety policy* in three parts:

1. A Floyd-style *verification-condition generator* (also referred to as the VC generator) [6], which is a procedure that computes a predicate in first-order logic based on the code to be certified. We will refer to this predicate as the *safety predicate*.
2. A set of axioms that can be used to validate the safety predicate.
3. The *precondition*, which is essentially a “calling convention” that defines how the code consumer will invoke the PCC binaries.

It is the job of the designer of the code consumer (e.g., the operating system designer) to define the safety policy. In practice, several different safety

policies might be used, each one tailored to the needs of specific tasks or services.

We obtain the VC generator by first specifying an *abstract machine* (also called the *operational semantics*), that simulates the execution of safe programs. The abstract machine is not strictly required but it simplifies the design of the safety policy and provides a basis for proving the soundness of the whole approach.

In order to make all of this more concrete, we will now present an example of an abstract machine that specifies a general form of memory safety for the DEC Alpha processor, and then show how the safety policy of a simple resource access service can be defined by a precondition. The VC generator and axioms will then be given in the next subsection.

An abstract machine for memory-safe DEC Alpha machine code

Because the experiments in this paper use the DEC Alpha assembly language, our abstract machine is essentially a high-level formal description of the Alpha architecture [19]. To see how this is done, consider the subset of the Alpha instruction set shown in Figure 2. (Actually, we use a larger subset of the DEC Alpha assembly language in our experiments, but this smaller subset will suffice for presentation purposes.) In this table, n denotes an integer constant and r_i refers to machine register i . All instructions operate on 64-bit values. For simplicity we allow the use of only 11 temporary and caller-save machine registers (which, for the purpose of this presentation, we rename r_0 through r_{10}). The consequence of this is that programs cannot write into reserved and callee-save registers (according to the standard C calling convention for the DEC Alpha architecture), and are thus trivially safe with respect to these registers.

To define how programs are executed, we define an abstract machine as a state-transition function, the essential core of which is shown in Figure 3. In this specification, the DEC Alpha program is a vector of instructions, Π , and the current instruction is Π_{pc} , where pc is the program counter. The variable ρ denotes the state of the machine registers and memory. The state-transition function maps a machine state (ρ, pc) into a new state (ρ', pc') by executing the current instruction Π_{pc} .

The notation $\rho[r_i]$ (often abbreviated as r_i) refers to the value of register r_i in state ρ .¹ The expres-

¹Valid register values are positive integers in the range 0 to $2^{64} - 1$. This constraint is expressed formally by the equation “ $r_i \bmod 2^{64} = r_i$ ”, which is applied to all register val-


```

op      ::= n | ri      i ∈ 0...10
al      ::= ADDQ | SUBQ | AND | OR | SLL | SRL
br      ::= BEQ | BNE | BGE | BLT
instr   ::= LDQ rd, n(rs) | STQ rs, n(rd) | al rs, op, rd | br rs, n | RET

```

Figure 2: The subset of DEC Alpha assembly language.

sion $\rho[r_d \leftarrow r_d \oplus 1]$ denotes the new state obtained from state ρ by incrementing the value of register r_d . So, for example, the Alpha “ADDQ r_s, op, r_d ” instruction is defined by Figure 3 to have the following semantics:

$$(\rho[r_d \leftarrow r_s \oplus op], pc + 1)$$

where ρ is the current register and memory state. This specification states that the ADDQ instruction updates register r_d with the sum of r_s and op , and also increments the program counter. We use the “circled” operation \oplus to denote two’s-complement addition on 64 bits. This operation is defined in terms of the usual integer arithmetic operations as

$$e_1 \oplus e_2 = (e_1 + e_2) \bmod 2^{64}$$

To model the state of memory, we use a pseudo register, called r_m , that gives the content of each memory location. We write $\text{sel}(r_m, a)$ for the contents of memory address a , and $\text{upd}(r_m, a, r_s)$ for the new memory state resulted from writing register r_s to address a . Memory operations work on 64-bits and the addresses involved must be aligned on an 8-byte boundary.

In the definition of the load and store instructions, there is a crucial difference between the DEC Alpha processor and our abstract machine. The difference is that our abstract machine performs the safety checks that are shown in boxes in Figure 3. For example, consider the definition of the “LDQ $r_d, n(r_s)$ ” instruction:

$$(\rho[r_d \leftarrow \text{sel}(r_m, r_s \oplus n)], pc + 1), \text{ if } \boxed{\text{rd}(r_s \oplus n)}$$

The predicate $\text{rd}(a)$ is true when it is safe to read the word at memory address a , which for the DEC Alpha implies that a is aligned on an 8-byte boundary. Similarly, the predicate $\text{wr}(a)$ is true when the address a denotes an aligned location that can be safely read or written. In essence, these checks define what is meant by safety, and more specifically for this example, memory safety. For the purpose of this paper, the predicates $\text{rd}(a)$ and $\text{wr}(a)$ are defined by the safety policy through the precondition, as shown in the next subsection.

ues. Negative values are represented using two’s-complement representation.

Mathematically, the abstract machine does not return errors when a $\text{rd}(a)$ or $\text{wr}(a)$ check fails. Instead, the execution blocks because there are no transition rules covering the error cases. In this setting, a program is safe if and only if it runs without blocking on the abstract machine. Of course, the presence of these safety checks means that the abstract machine is not a faithful abstraction of the DEC Alpha processor. However, the purpose of certification is to prove that all safety checks always succeed. If we have a valid safety proof for a program, we know that we can safely execute it on a real DEC Alpha and get the same behavior as on our abstract machine, even though the Alpha does not implement the safety checks.

There are other notable differences between our abstract machine and a real DEC Alpha. For example, to simplify the presentation in this paper, we have restricted all branches to be only forward. Allowing backward branches and loops introduces a number of complications, but is handled in a conceptually straightforward manner through the addition of explicit loop invariants. As it turns out, the packet filter examples we use in our experiments do not have any loops, and so it is not inconvenient to eliminate them here. In a later section we will briefly describe our experiments with looping programs, including a safe IP-header checksum routine.

Another interesting aspect of the abstract machine is the level of abstraction of our specification. We might try to be ambitious and make a complete specification of the DEC Alpha processor. However, this would be extremely complex and probably difficult to trust. And, as a practical matter, for specific tasks such as the ones we are considering, many details and features of the Alpha are irrelevant. This justifies working at a higher level of abstraction above the details of the pipeline, cache, timing, and interrupt behavior.

We can also consider encoding other kinds of safety checks into our abstract machine. For the sake of simplicity, we have specified only a notion of fine-grained memory safety. With some ingenuity, an abstract machine designer can define safety policies involving other kinds of safety, like control over resource usage or preservation of data-abstraction

$$(\rho, pc) \rightarrow \begin{cases} (\rho[r_d \leftarrow r_s \oplus op], pc + 1), & \text{if } \Pi_{pc} = \text{ADDQ } r_s, op, r_d \\ (\rho[r_d \leftarrow \text{sel}(r_m, r_s \oplus n)], pc + 1), & \text{if } \Pi_{pc} = \text{LDQ } r_d, n(r_s) \text{ and } \boxed{\text{rd}(r_s \oplus n)} \\ (\rho[r_m \leftarrow \text{upd}(r_m, r_d \oplus n, r_s)], pc + 1), & \text{if } \Pi_{pc} = \text{STQ } r_s, n(r_d) \text{ and } \boxed{\text{wr}(r_d \oplus n)} \\ (\rho, pc + n + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \text{ and } r_s = 0 \\ (\rho, pc + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \text{ and } r_s \neq 0 \end{cases}$$

Figure 3: The Abstract Machine.

$$VC_{pc} = \begin{cases} VC_{pc+1}[r_d \leftarrow r_s \oplus op], & \text{if } \Pi_{pc} = \text{ADDQ } r_s, op, r_d \\ \text{rd}(r_s \oplus n) \wedge VC_{pc+1}[r_d \leftarrow \text{sel}(r_m, r_s \oplus n)], & \text{if } \Pi_{pc} = \text{LDQ } r_d, n(r_s) \\ \text{wr}(r_d \oplus n) \wedge VC_{pc+1}[r_m \leftarrow \text{upd}(r_m, r_d \oplus n, r_s)], & \text{if } \Pi_{pc} = \text{STQ } r_s, n(r_d) \\ (r_s = 0 \Rightarrow VC_{pc+n+1}) \wedge (r_s \neq 0 \Rightarrow VC_{pc+1}), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \\ \text{Post}, & \text{if } \Pi_{pc} = \text{RET} \end{cases}$$

Figure 4: The Verification-Condition Generator.

boundaries. Once a safety policy is defined, application writers are free to use it to create PCC binaries that guarantee safety.

A sample application and its precondition

The abstract machine as given above describes safety in terms of the abstract notions of readable and writable memory locations. For this to be useful, the code consumer must specify an interface to PCC binaries that identifies the readable and writable memory locations. We do this by specifying a *precondition*, which is a predicate in first-order logic that the code consumer guarantees to be valid when the PCC binary is invoked.

Consider the following simple example. Suppose an operating-system kernel maintains an internal table with data pertaining to various user processes. Each table entry consists of two consecutive memory words—a tag and a data word. The tag describes whether the data word is user writable or not. The kernel also provides a *resource access service* through which user processes are given permission to access their table entry by installing native code in the kernel. To make this possible the kernel invokes the user-installed code with the address of the table entry corresponding to the parent process in machine register r_0 . This address is guaranteed by the kernel to be valid and aligned on an 8-byte boundary.

Although this example is somewhat contrived, we can imagine that entries in the table represent capabilities (perhaps file descriptors), and so we would

like to provide user-installed code with full access to the correct table entries, while maintaining the integrity of the rest of the table and other parts of the kernel state.

Informally, the safety policy for the resource access service requires that: (1) the user code cannot access other table entries besides the one pointed to by r_0 , (2) the tag is read only, (3) the data word is also read only unless the tag value is non zero, and, (4) the code does not modify reserved and callee-saves registers. The last condition ensures that the kernel can safely invoke the user code using a normal C function call.

More formally, the kernel specifies a precondition Pre_r , which states that it is safe to read the tag pointed to by r_0 , and that it is also safe to write the data at offset 8 from r_0 if the contents of the tag is not 0. In formal notation, this is written as follows:

$$Pre_r = r_0 \bmod 2^{64} = r_0 \wedge \text{rd}(r_0) \wedge \text{rd}(r_0 \oplus 8) \wedge \text{sel}(r_m, r_0) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8)$$

What remains now is to prove for a particular client of the resource access service that all $\text{rd}(a)$ and $\text{wr}(a)$ checks will always succeed, given this precondition and abstract machine. In general, we can also specify a postcondition as part of the safety policy, which would require particular invariants to be valid when the user code terminates. Conceptually, in our example the postcondition is the predicate **true**, meaning that no additional conditions are imposed on the final machine state.

Before moving on to a discussion of the proof

generation process, we note that the safety policy we have described here can be thought of as enforcing fine-grained memory protection. In general, one could imagine having much more involved safety requirements. For example, we could change the tag word in the table entry to be a semaphore that the user code must acquire (e.g., atomically test-and-set to zero) before trying to write the data word; furthermore, we could also require (via a simple postcondition) that the code releases the semaphore before returning. Again, for purposes of the current presentation, we stick to the simpler memory-safety requirements.

2.2 Certifying the Safety of Programs

To create safety proofs for a program, we must prove that executing it does not violate any of the safety checks (and the postcondition, if one is given, is also satisfied). Standard techniques exist for building such proofs. Our technique is based on Floyd's verification conditions [6], because they are powerful enough to deal with unstructured assembly-language programs and a broad range of safety invariants. Similar techniques have been used before to verify assembly-language programs [2, 3].

Certification of programs involves two steps:

1. Compute the *safety predicate* for the program. This essentially encodes the semantic meaning of the program in logical form and constitutes a formal statement that the program, when executed, will not violate any safety checks.
2. Generate a *proof* of the safety predicate, written out in a checkable form.

Both these steps are described in the following subsections.

Computing the safety predicate

To compute the safety predicate, we first generate a vector VC of predicates, one for each instruction as specified by the rules in Figure 4. The notation VC_{pc} denotes the predicate for the current instruction. Since the rules specify VC_{pc} in terms of VC_{pc+1} , the verification-condition VC_0 for the beginning of the program can be computed by starting at the end of the program and working back towards the beginning.²

²This simple approach works because all branches are restricted to be forward-only. We discuss later what happens in the presence of loops.

The rules in Figure 4 are derived in a straightforward manner from the abstract machine specification of Figure 3; in fact, we imagine that experienced kernel and safety policy designers would skip the abstract machine specification and give only the VC generator rules. The notation $P[r_d \leftarrow r_s \oplus op]$ stands for the predicate obtained from P by substituting $r_s \oplus op$ for r_d .

After computing the vector VC , the safety predicate is computed simply by plugging the program Π , precondition Pre , and postcondition $Post$ into the following formula:

$$SP(\Pi, Pre, Post) = \forall r_0 \dots \forall r_{10} \forall r_m. Pre \Rightarrow VC_0$$

The intuition behind a valid safety predicate is that for any initial state that satisfies the precondition Pre , the code Π starting at the first instruction executes without failure and, if it terminates, the final state satisfies the postcondition $Post$.

			%Address of tag in r_0
1	ADDQ	$r_0, 8, r_1$	%Address of data in r_1
2	LDQ	$r_0, 8(r_0)$	%Data in r_0
3	LDQ	$r_2, -8(r_1)$	%Tag in r_2
4	ADDQ	$r_0, 1, r_0$	%Increment Data in r_0
5	BEQ	r_2, L_1	%Skip if tag == 0
6	STQ	$r_0, 0(r_1)$	%Write back data
L_1	RET		%Done

Figure 5: DEC Alpha assembly code for resource access. Initially register r_0 holds the address of the tag. The data is at the offset 8 from r_0 .

For a concrete example of client code for the resource access service, consider the small program in Figure 5. The overall effect of this program is to increment the data word if it is writable. We first compute VC_0 for this program using the rules in Figure 4; then we compute the safety predicate SP_r using the above formula with the precondition Pre_r and the postcondition $true$. After a few trivial simplifications, the resulting safety predicate is the following:

$$SP_r = \forall r_0. \forall r_m. Pre_r \Rightarrow rd(r_0 \oplus 8) \wedge rd(r_0 \oplus 8 \oplus 8) \\ \wedge sel(r_m, r_0 \oplus 8 \oplus 8) = 0 \Rightarrow true \\ \wedge sel(r_m, r_0 \oplus 8 \oplus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)$$

Informally, the SP_r predicate says that for all values of register r_0 and states of memory r_m satisfying the precondition Pre_r , the memory locations $r_0 \oplus 8$ and $r_0 \oplus 8 \oplus 8$ must be readable and if the tag (at address $r_0 \oplus 8 \oplus 8$) is non zero, the data (at address $r_0 \oplus 8$) must be writable. All these conditions must be true for the code to be safe with respect to the resource access safety policy.

Proving the safety predicate

We have intentionally written the program in Figure 5 in a slightly complicated way, to show that low-level optimizations do not pose significant problems in generating and validating safety proofs. Three of the interesting properties of this program are (1) the instructions are somewhat scheduled, including speculative execution of the load in line 2 and of the addition in line 4, to accommodate the DEC Alpha pipeline latency³, (2) register r_0 is reused in line 2 to hold the data word instead of the tag address, and (3) even though the precondition is expressed as a function of the value in register r_0 , some of the actual memory accesses are done through register r_1 . In general, we expect scheduling and register allocation to have no effect on the safety predicate and its proof.

It is a simple exercise for the reader familiar with assembly-language programming to verify that this code is indeed correct with respect to the safety policy. The problem, of course, is how to convince even the most suspicious kernel that this code is absolutely safe. To do this, we must prove the safety predicate according to the rules of first-order predicate calculus extended with two's-complement integer arithmetic. We refer to this set of proof rules as Σ and we write $\vdash_{\Sigma} SP$ when the safety predicate SP can be proved according to the rules in the set Σ . Most of the rules in Σ are simple. Below we show two of the rules we use, the first being a classical implication-elimination rule from the predicate calculus, and the second a rule about arithmetic:

$$\begin{array}{ll} \vdash_{\Sigma} Q, & \text{if } \vdash_{\Sigma} P \Rightarrow Q \text{ and } \vdash_{\Sigma} P \\ \vdash_{\Sigma} e_1 \oplus e_2 \ominus e_2 = e_1, & \text{if } \vdash_{\Sigma} e_1 \bmod 2^{64} = e_1 \end{array}$$

The second rule is perhaps a bit surprising because $e_1 + e_2 - e_2 = e_1$ is unconditionally true in integer arithmetic. However, for the machine implementation of arithmetic, this statement is true only if the original value of e_1 is a valid register value.

A large fragment of the proof of the safety predicate for our example program is shown in a proof-tree form in Figure 6. This proof was generated automatically by our PCC system, which incorporates a simple theorem prover. We use vertical dots to stand for extractions of a conjunct from the precondition. You can read the proof tree from top to bottom, interpreting every node as a valid inference of the predicate below the line using the assumptions above the line. For example, in the upper-right corner of the figure the predicate $r_0 = r_0 \oplus 8 \ominus 8$ is

³These operations are speculative because they are not required if the branch in line 5 is taken.

proved using the arithmetic rule we discussed with the assumption $r_0 \bmod 2^{64} = r_0$ extracted from the precondition. Then $\text{wr}(r_0 \oplus 8)$ is proved using the implication-elimination rule and the hypothesis u of the predicate $\text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0$. This hypothesis is introduced at a lower level in the proof tree, at the node labeled u , for the purpose of proving the predicate $\text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8)$.

The guarantee of safety

We use the proof of the safety predicate, written out in an appropriate language (to be described in the next section), as the proof that the code obeys the safety policy. This is justified formally by the *safety theorem*, stated below:

Theorem 2.1 (Safety) *For any program Π , precondition Pre and postcondition $Post$, if $\vdash_{\Sigma} SP(\Pi, Pre, Post)$ then for any initial state ρ_0 that satisfies the precondition and for any abstract machine state (ρ, pc) originating from the initial state $(\rho_0, 0)$, one of the following is true:*

1. *The state (ρ, pc) is a final state (i.e. $\Pi_{pc} = \text{RET}$) satisfying the postcondition $Post$, or*
2. *The execution is not stuck, i.e., there exists a new state (ρ', pc') such that $(\rho, pc) \rightarrow (\rho', pc')$.*

Since the abstract machine gets stuck when there is any violation of an $\text{rd}(a)$ or $\text{wr}(a)$ safety check, this theorem provides an absolute guarantee that a certified program will not have such violations, as long as its execution is started in a state that satisfies the precondition.

The proof of the Safety Theorem is beyond the scope of this paper, but can be found in a separate technical report [16].

2.3 Validating the Safety Proofs

A PCC binary consists of the assembled native code together with an encoding of the proof of its safety predicate. To validate the binary, the code consumer first extracts the native code and then computes its safety predicate using the VC rules. Then, it checks that the safety proof is a valid proof of the safety predicate.

This method ensures safety even if the native code or the proof in the PCC binary is tampered with. If the code is modified, then in all likelihood its safety predicate changes, so the given proof will not correspond to it. If the proof is modified, then either it will be invalid, or else not correspond to the safety predicate. If the code is modified in such

$$\begin{array}{c}
\begin{array}{c} \vdots \\ \text{Pre}_r \\ \vdots \\ \text{rd}(r_0) \end{array} \quad \begin{array}{c} \vdots \\ \text{Pre}_r \\ \vdots \\ r_0 \bmod 2^{64} = r_0 \end{array} \quad \begin{array}{c} \vdots \\ \text{Pre}_r \\ \vdots \\ \text{sel}(r_m, r_0) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8) \end{array} \quad \begin{array}{c} u \\ \text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \end{array} \quad \begin{array}{c} \vdots \\ \text{Pre}_r \\ \vdots \\ r_0 \bmod 2^{64} = r_0 \\ r_0 = r_0 \oplus 8 \ominus 8 \end{array} \\
\hline
\text{rd}(r_0 \oplus 8 \ominus 8) \quad \text{wr}(r_0 \oplus 8) \quad \text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8) \quad \dots \\
\hline
\text{rd}(r_0 \oplus 8 \ominus 8) \wedge (\text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8)) \wedge \dots \quad \text{Pre}_r \\
\hline
\text{Pre}_r \Rightarrow \text{rd}(r_0 \oplus 8 \ominus 8) \wedge (\text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8)) \wedge \dots \\
\hline
\forall r_0. \forall r_m. \text{Pre}_r \Rightarrow \text{rd}(r_0 \oplus 8 \ominus 8) \wedge (\text{sel}(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \text{wr}(r_0 \oplus 8)) \wedge \dots
\end{array}$$

Figure 6: A Fragment of the formal safety proof of SP_r .

a way that the safety predicate is unchanged (for example, instruction scheduling and register allocation might do this in typical circumstances), or if both the code and the proof are modified so that we still have a valid proof of the new safety predicate, the validation succeeds and we continue to retain a guarantee of safety.

To automate the validation process, we must first choose a concrete representation language for predicates and their proofs. From the many available choices, we have selected the Edinburgh Logical Framework [7] (also called LF) as the representation framework for predicates and proofs. LF is an extension of the simply typed lambda calculus and was designed as a meta language for high-level specification of languages in logic and computer science. The most attractive property of LF is that it has a powerful yet simple typechecking algorithm, which we use to check the validity of proofs.

We represent the predicates and the proofs in LF in such a way that the validity of a proof is implied by the well typedness of the proof representation. Thus, proof validation amounts to typechecking. Also, LF allows us to represent in an elegant way a few key issues in logical proof correctness, such as the manipulation of logical parameters and assumptions. It is well beyond the scope of this paper to discuss in detail LF and the typechecking algorithm, however it is worth mentioning that typechecking is decidable and is described by a few simple rules. Indeed, typechecking is so simple that any programmers who do not trust the publicly available implementation can implement it easily themselves. Our implementation has about five pages of C code, even though it incorporates a few optimizations to the basic algorithm. With this implementation, it takes 1.4 milliseconds to validate the proof of the

SP_r predicate.

For flexibility and to allow easy exchange of proofs between system components, we have designed a binary encoding of LF representations. Thus, a typical PCC binary contains a section with the native code ready to be mapped into memory and executed, followed by a symbol table used to reconstruct the LF representation at the code consumer site, and the binary encoding of the LF representation of the safety proof. The latter component is the safety proof. Figure 7 shows the sizes of these sections for the PCC binary corresponding to the resource access example.

NATIVE CODE SECTION	0
RELOCATION SECTION	45
PROOF SECTION	220
	340

Figure 7: The layout of the PCC binary for the resource access example. The offsets are in bytes.

Currently, PCC binaries for standard packet filters, including the native code, safety proof, and relocation section, are about 400 to 1200 bytes in size, with the proof about 3 times larger than the code. The size of the relocation section increases linearly with the number of distinct proof rules used in the proof. In the case of packet filter safety proofs, the relocation section is a third of the binary but we expect this ratio be much smaller for larger proofs.

There is a considerable amount of design latitude in the encodings of the proofs, and we have barely scratched the surface on what can be done to reduce the size of the binaries as well as the time required for validation. But already, with relatively little effort, we have achieved acceptably small binaries and low validation times.

3 Application: Network Packet Filters

In order to gain more experience with PCC and to compare it with other approaches to code safety, we have performed a series of experiments with safe network packet filters. We describe in this section the particulars of the PCC approach to network packet filters. Then in Section 3.1, we compare it with other approaches including interpreted packet filters (as exemplified by the BSD Packet Filter), code editing (through Software Fault Isolation), and using a safe programming language (the approach taken in the SPIN kernel).

A packet filter is an application-provided subroutine that scans each incoming network packet and decides whether the user application is interested in receiving it or not. Packet filters are supported by most of today's workstation operating systems. Since their first introduction in [15], packet filters have been used successfully in network monitoring and diagnosis.

In the PCC approach the packet filter is a PCC binary whose native code component is invoked by the kernel on each incoming network packet. Kernel safety is ensured by validating the safety proof.

Following the procedure described in Section 2 we first establish a safety policy. To allow for a fair comparison we follow the BSD Packet Filter model of safety. The packet filter code can examine the packet at will and can also write to a statically allocated scratch memory. Informally, the safety policy requires that: (1) memory reads are restricted to the packet and the scratch memory; (2) memory writes are limited to the scratch memory; (3) all branches are forward; and (4) reserved and callee-saves registers are not modified. These rules establish memory safety and termination assuming that the kernel calls the packet filter with valid packet and scratch memory addresses.

We write the packet filter code assuming that the return value must be in r_0 , the aligned address and the length of the packet filter are given in r_1 and r_2 respectively, and the address of a 16-byte aligned scratch memory is given in r_3 . Moreover the packet's length is positive and at least 64-bytes (the mini-

mum length of an Ethernet packet). Formally this is expressed as the precondition:

$$\begin{aligned} Pre = & \quad r_1 \bmod 2^{64} = r_1 \wedge \\ & \quad r_2 \bmod 2^{64} = r_2 \wedge r_2 < 2^{32} \wedge r_2 \geq 64 \wedge \\ & \quad r_3 \bmod 2^{64} = r_3 \wedge \\ & \quad \forall i. (i \geq 0 \wedge i < r_2 \wedge (i \& 7) = 0) \\ & \quad \quad \Rightarrow rd(r_1 \oplus i) \quad \wedge \\ & \quad \forall j. (j \geq 0 \wedge j < 16 \wedge (j \& 7) = 0) \\ & \quad \quad \Rightarrow wr(r_3 \oplus j) \quad \wedge \\ & \quad \forall i. \forall j. (i \geq 0 \wedge i < r_2 \wedge j \geq 0 \wedge j < 16) \\ & \quad \quad \Rightarrow (r_1 \oplus i \neq r_3 \oplus j) \end{aligned}$$

The first few conjuncts of the precondition restrict the values of input registers to valid machine word values. The last term of the precondition rules out the possibility of memory aliasing between packets and the scratch memory. This is useful when reasoning about filters that write to the scratch memory.

The postcondition in our packet filter experiment is the predicate **true**, meaning that no additional conditions are placed on the final state.

We have implemented four typical packet filters in assembly language and certified their safety with respect to the packet filter safety policy. Filter 1 accepts all IP packets. This is done by comparing a 16-bit word in the packet to a given value. Filter 2 accepts IP packets originating from a given network. This involves checking a 24-bit value in addition to the work done by Filter 1. Filter 3 accepts IP or ARP packets exchanged between two given networks. This includes all the work done by Filter 2 with the addition of checking the destination network address. Extra complexity is required because of different header layout of IP and ARP packets. Filter 4 accepts all TCP packets with a given destination port. This filter has to check that the Ethernet packet is an IP packet, then that it is a TCP packet, and lastly that the destination port matches a given value. The offset of the TCP destination port is computed based on a byte in the IP header (the length of the IP header).

The effort involved in hand-coding packet filters in assembly language is repaid in increased performance, because packet filters are usually small and very frequently executed. Hand-coding provides the opportunity to perform optimizations that are difficult to obtain from an optimizing compiler. The important point is that these optimizations are not an impediment to generation and validation of safety proofs. Here are a few optimizations that we incorporated in our packet filters:

- The number of memory operations is minimized by using the DEC Alpha 64-bit load fol-

lowed by byte extraction.

- The TCP port number can be found at packet offset $(([14]_8 \& 15) * 4 + 16)$, where $[14]_8$ denotes the byte at offset 14. If loading 64 bits at a time on a little-endian machine, the formula becomes $((([8]_{64} \gg 48) \& 255) \& 15) * 4 + 16$. With further simplification we reduce this to $(([8]_{64} \gg 46) \& 60) + 16$, which is exactly how we coded Filter 4.

After we write a packet filter, our prototype assembler produces its safety predicate using the verification-condition method presented in Section 2. The safety predicate is then proved using a theorem prover. We currently use our own theorem prover, which is admittedly a toy. When it gets stuck, it requires intervention from the programmer, mainly to learn new axioms about arithmetic (for example, to know that $r_1 > 0 \Rightarrow r_1 \geq 0$). The process is easy, and because user-provided axioms are remembered for future sessions, by now our system works automatically for most practical packet filters. With state-of-the-art theorem proving technology we expect to be able to prove completely automatically most arithmetic facts involved in certifying packet filters.

With our primitive theorem-prover we can generate safety proofs for packet filters in about 5 to 10 seconds, in the cases when no user-intervention is required.

3.1 Performance Comparisons

All performance measurements were done on a DEC Alpha 3000/600 with a 175-MHz processor, a 2-MByte secondary cache and 64-MByte main memory, running OSF/1. All measurements were performed off-line using a 200,000-packet trace from a busy Ethernet network at Carnegie Mellon University.

We measured the average per-packet run time of the four PCC packet filters and of functionally equivalent filters implemented using alternative approaches: the BSD Packet Filter architecture, Software Fault Isolation and programming in the safe subset of Modula-3. In our experiments with Modula-3 packet filters we use the VIEW extension [9] for pointer-safe casting. The result of the measurements are shown in Figure 8. From a per-packet latency point of view, the PCC packet filters outperform filters developed using any other considered approach. However, the PCC method has a startup cost significantly larger than the other approaches. This cost is the proof validation time,

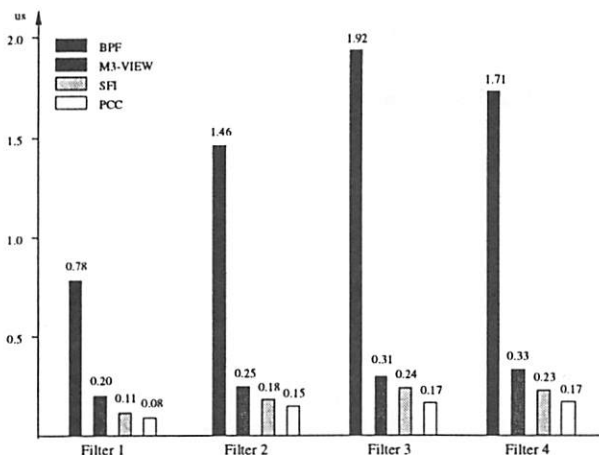


Figure 8: Comparison of average per-packet run time.

which is presented in Table 1 together with the PCC binary size for all four filters and maximum heap space used for validation. The maximum depth of the stack during validation was under 4 Kbytes.

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Binary Size (bytes)		385	516	1024	814
Validation Cost	Time (μ s)	780	1070	2350	1710
	Space (KB)	5.5	8.7	24.6	15.1

Table 1: Proof size and validation cost for PCC packet filters.

Despite the relatively high validation cost, the run-time benefits of PCC packet filters are large enough to amortize the startup cost after processing a reasonable number of packets. Figure 9 shows the overall running time, including startup cost, as a function of the number of packets processed, for Filter 4. In this particular case, the cost of proof validation is amortized after 1200 packets when compared to the BPF version of the filter, after 10500 packets when compared to the Modula-3 version and after 28,000 packets when compared to the SFI packet filter. Note that at the time we collected the packet trace used for the experiments we counted about 1000 Ethernet packets per second on the average.

We proceed now to describe in more detail each considered approach focusing on how it relates to PCC from the safety point of view, and how we set up the performance measurements.

The standard way to ensure safe execution of packet filters is to interpret the filter and perform extensive run-time checks. This approach is best exemplified by the BSD Packet Filter architecture [13],

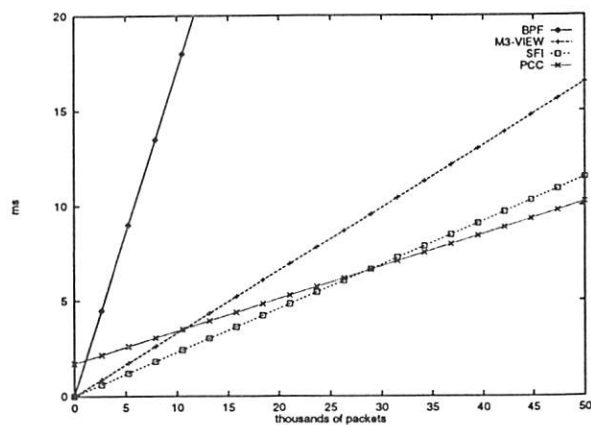


Figure 9: Startup cost amortization for Filter 4.

commonly referred to as BPF. In the BPF approach the filter is encoded in a restricted accumulator-based language. According to the BPF semantics, a filter that attempts to read outside the packet or the scratch memory, or to write outside the scratch memory, is terminated and the packet rejected.

The BPF interpreter makes a simple static check of the packet filter code to verify that all instruction codes are valid and all branches are forward and within code limits. We measured this one-time overhead to be a few microseconds, which is negligible. BPF packet filters, however, are about 10 times slower than our PCC filters. In the PCC approach all checks are moved to the validation stage, allowing for much faster execution.

In order to collect data for the BPF packet filters, we extracted the BPF interpreter as implemented by the OSF/1 kernel and compiled it as a user library.

It is possible, of course, to eliminate the need for interpretation. For example, we could replace the packet-filter interpreter with a compiler. This approach is taken by several researchers [10, 24]. The problem here is the startup cost and complexity of compilation, especially if serious optimizations are performed.

Another approach to safe code execution is Software Fault Isolation (SFI) [23]. SFI is an inexpensive method for parsing binaries and inserting run-time checks on memory operations. There are many flavors of SFI depending on the desired level of memory safety. If the entire code runs in a single protection domain whose size is a power of 2, and if only memory writes are checked, then the run-time cost of SFI is relatively small. If, on the other hand, the untrusted code interacts frequently with the code consumer or other untrusted components residing in different protection domains and the read operations

must be checked also, the overhead of the run-time checks can amount to 20% [23]. A more serious disadvantage of SFI is that it can only ensure memory safety. We believe that this level of safety is not enough in general, and that it is important to be able to check abstraction boundaries and representation invariants, as shown by the resource access example in Section 2.

In order to accommodate SFI for packet filters, we allowed some concessions to the packet filter semantics. For example, we assumed that the kernel allocates the packets on a 2048-byte boundary. Furthermore, we assume that the filter can safely access the entire segment of 2048 bytes, independently of the packet size. Note that the BPF packet filter semantics, which we followed for all other experiments, specifies that a filter should be terminated if it tries to access beyond the packet boundary. This means that some working packet filters in the BPF semantics will not behave as expected in the SFI semantics for packet filters, and vice-versa.

One common way of performing SFI is at the code producer site, usually as part of the code-generation phase in a compiler. In this case, the code consumer performs a load-time checking that SFI was done correctly. The load-time SFI validator is reportedly simple if it must deal only with binaries for which run-time checks have been inserted on every potentially dangerous memory operation [23]. On the other hand, in the case where the validator must accept binaries for which the number of run-time checks has been optimized through program analysis, the validator itself has to redo the analysis that led to the optimization. This means a more complex and slower validation, and in fact such an SFI validator does not presently exist.

We inserted run-time checks for the memory operations in the assembly language packet filters implemented for the PCC experiment. This process can be done by a simple and easy-to-trust implementation of SFI. In our experiments, PCC packet filters run about 25% faster than SFI filters.

As part of our SFI experiment, we produced safety proofs attesting that the resulting SFI packet filter binaries are safe with respect to the packet filter safety policy. We achieve the same effect as an SFI load-time validator but using the universal type-checking algorithm and a few application-dependent proof rules. The precondition for this experiment says that it is safe to read from any aligned address that is in the same 2048-byte segment with the packet start address. Proof sizes and validation times are very similar to those for plain PCC packets.

Another approach to safe code is to use a type-safe programming language. This approach is taken by the SPIN extensible operating system [1], and the language used is Modula-3 [17] extended with pointer-safe casting (VIEW). SPIN allows applications to install extensions in the kernel but only if they are written in the safe subset of Modula-3. The extensions are compiled by a trusted compiler and the resulting executable code is then believed to be safe (at least according to the Modula-3 model of safety). Note that such extensions written in Modula-3 are intrinsically safe, as anyone who believes in the safety of Modula-3 can check their compliance with Modula-3 syntactic and typing rules.

We believe that encoding kernel extensions as PCC binaries instead of Modula-3 source code can provide important benefits. One such benefit is the increased flexibility for extension writers because any native code extension can be accepted, independent of the original source language or even the compiler used, as long as a valid safety proof accompanies it. Another potential benefit is overcoming the limitations of the Modula-3 safety model: the PCC safety proof should be able to express properties such as disciplined use of locks or array bounds compliance with no need for run-time checks.

We wrote the four packet filters in the safe subset of Modula-3 and compiled them with the version 3.5 of the DEC SRC compiler extended with the VIEW operation [24]. VIEW is used to safely cast the packet filter to an array of aligned 64-bit words allowing fewer memory operation for accessing packet fields. In contrast, in plain Modula-3 the packet fields must be loaded a byte at a time, and a safety bounds check is performed for each such operation. The compiler tries to eliminate some of these checks statically but it is not very successful for packet filters. The main reason is that a critical piece of information, the fact that packets are at least 64 bytes long, cannot be communicated to the compiler through the Modula-3 type system.

We measured a 20% improvement in the Modula-3 packet filter performance when using VIEW. Similar performance improvements over the DEC SRC Modula-3 compiler have been reported [18] for the more recent Vortex compiler. However, since we have not conducted any experiments with the Vortex compiler on our packet filters, it is not clear what kind of improvements we would realize in practice.

In an alternate implementation of untrusted code certification using Modula-3, the source code is compiled by a trusted and secure compiler that signs the executable for future use. Validation then means cryptographic signature checking and like in the

PCC approach there is no run-time cost associated with it. We do not have a complete implementation of such a cryptographic validation, so we do not know exactly how large is the startup cost for the digital signature approach. It is likely however that a good implementation of digital signatures would achieve faster validation and significantly faster generation of certificates. The essential drawback of cryptographic techniques over PCC is that validation establishes only a trusted origin of the code and not its absolute safety relative to the safety policy. In particular, a digital signature can be ascribed to an unsafe program just as easily as to a safe one. Also, the cost of managing and transmitting encryption keys is not incurred by PCC.

We should mention here one more approach to safe code execution, although we do not have an actual quantitative comparison. The Java Virtual Machine [21] is a proposed solution to safe interaction of distributed, untrusted agents. Mobile code is encoded in the Java Virtual Machine Language (also referred to as Java Bytecode), which is basically a safe low-level imperative language. Safety is achieved through a combination of static typechecking and run-time checking.

However, the Java Bytecode safety model is relatively limited as a result of limitations of the type system. For example the Java Bytecode type information encoded in the instruction codes can only express a few basic abstract types (e.g., integers, objects) and has no provisions for expressing safety policies like the one for the resource access example in Section 2. Also, invariants involving array bounds compliance cannot be expressed in the Java Bytecode type system and must be checked at run time.

Although Java Bytecode is a low-level language, it still requires substantial processing before it can be executed on a general-purpose processor. In contrast, PCC segregates the safety proof from the program code, allowing for the code portion to be encoded in a variety of languages, including native code, without any safety loss.

4 Practical Problems and Future Work

In order to create a safety proof, the code producer must prove a predicate in first-order logic. In general, this problem is undecidable. However, as we mentioned in Section 1, the code producer can resort to "extra" run-time checks inserted in strategic locations, which have the tendency to simplify the certification.

Fortunately, in the packet-filter experiments, the certification process is nearly automatic, and we have not been forced to insert any extra run-time checks into the code. In fact, we find that safety predicates for packet filters are fairly easy handled by existing theorem-proving technology.

One of the simplifications in the packet filters is to restrict programs so that they do not contain loops. Although the general framework presented in this paper is easily extended to accommodate loops [5], this introduces a number of complications. One experiment we conducted involves an IP-header checksum routine, which is hand-coded in 39 DEC Alpha instructions. The core loop contains 8 instructions, and is optimized by computing the 16-bit IP checksum using 64-bit additions followed by a folding operation. The resulting PCC binary for this routine is, as expected, quite fast, beating the standard C version in the OSF/1 kernel by a factor of two. The PCC binary itself is 1610 bytes in size and proof validation takes 3.6 milliseconds.

This experiment brought to light several complications. For example, the standard approach of verifying loops using Floyd-style verification conditions involves introducing loop invariants explicitly, which is a challenge for any theorem-proving technology and often requires user intervention. In fact, for general assembly-language programs this represents the most important problem to be solved, as it is the main obstacle in automating the generation of proofs. Since this is beyond the capabilities of our system, we are forced to write the invariants out by hand. This also means that the native code must be accompanied by a loop invariant for every loop. Thus, the PCC binary contains a mapping between each loop and its invariant. Our convention is to have the PCC binary contain a table that maps each backward-branch target to a loop invariant.

Besides the problem of how to generate the proofs, there is also the matter of their size. In principle, the proofs can be exponentially large (in the size of the program). This has not been a problem for any of the examples we have tried thus far, however. The blowup would tend to occur in programs that contain long sequences of conditionals, with no intervening loops. Perhaps we have not yet seen the problem in a serious way because such programs tend to be hard for humans to understand, and we are writing the programs by hand. But as a general matter, the size of the PCC binaries is an issue that must be addressed carefully. We have implemented several optimizations in the representation of the proofs, and much more is possible here. But ultimately, we need more practical experience to

know if this is a serious obstacle for PCC in practice.

For programs with loops, the loop invariants break a program with cycles into a set of acyclic code fragments. We treat each code fragment as a separate program, using the invariants as preconditions for each. This has the beneficial effect of partitioning the safety predicate and its proof into smaller pieces, and overall tends to reduce the size of the proof dramatically. For this reason, even for sections of programs that do not contain loops, it may be beneficial to introduce invariants, as a way of controlling the growth of the PCC binaries.

In addition to developing better certification technology, we see several other interesting directions for further research. One possibility that we intend to explore is the application of PCC to more dynamic properties, such as resource-usage guarantees. One example would be to certify that specific synchronization locks are always released prior to some action. The framework we have presented in this paper is already expressive enough to define such safety policies, and so what remains now is to try some experiments.

Another possibility is to allow the consumer and producer to “negotiate” a safety policy at run time. This would work by allowing the producer to send an encoding of a proposed safety policy (including the VC-generation rules, proof rules, and preconditions) to the consumer. If the consumer determines that the proposed policy implies some basic notion of safety, then it can allow the producer to produce PCC binaries using the new policy. This might be useful in distributed systems, in which one agent wants to define a language and then transmit to other agents code written in that language.

Finally, we believe there would be advantages to starting with a safe programming language and then implementing a *certifying compiler* that produces PCC binaries as target programs. For the safety properties that are implied by the source language, construction of the proofs is, in principle, a matter of having the compiler prove the correctness of the translation to target code. We have already experimented with a toy compiler of this sort for a small type-safe programming language, and hope to expand on this in the near future.

5 Conclusions

We have described *proof-carrying code*, a mechanism that allows a kernel or server to interact safely with binaries supplied by an untrusted source. PCC does not incur any run-time overhead for the kernel. In-

stead, the code producer is required to generate a formal proof that the code obeys the safety policy. The kernel can easily check the proofs for validity, after which it is absolutely certain that the code respects the safety policy. Furthermore, PCC binaries are completely tamper-proof; any attempt to alter either the native code or safety proof in a PCC binary is either detected or harmless. Our experiments with network packet filters show that PCC can lead to significant performance advantages over existing approaches to safe code, including code-editing techniques such as Software Fault Isolation.

Proof-carrying code has the potential to free the system designer from relying on run-time checking as the sole means of ensuring safety. Traditionally, system designers have always viewed safety simply in terms of memory protection, achieved through the use of rather expensive run-time mechanisms such as hardware-enforced memory protection and extensive run-time checking of data. By being limited to memory protection and run-time checking, the designer must impose substantial restrictions on the structure and implementation of the entire system, for example by requiring the use of a restricted application-kernel interaction model (such as a fixed system call or application-program interface.)

Proof-carrying code, on the other hand, allows the safety policy to be defined by the kernel designer and then certified by each application. Not only does this provide greater flexibility for designers of both the system and applications, but also allows safety policies to be used that are more abstract and fine-grained than memory protection. We believe that this has the potential to lead to great improvements in the robustness and end-to-end performance of systems.

6 Final Thoughts

The inspiration for proof-carrying code comes from the realm of static type systems, especially as embodied by the language Standard ML (SML). In the formal definition of SML [14], a formal theorem guarantees the safety of any type-correct SML program, for a rigorously defined notion of safety. There are, of course, many other type-safe programming languages, for example Modula-3 [17] and Java [20], but the use of mathematical formalism sets SML apart from these languages, and as a practical matter this rigor provides the basic conceptual and technical foundations that we need to create checkable proofs.

With type-safe languages like SML in mind, we can get an intuitive idea about how proof-carrying

code works. Consider a compiler for SML. Agent *A* writes an SML program and compiles it to a native-code target program. If we then throw away the source program, how can we later convince an agent *B* that the target program is safe? (We are assuming that agent *B* does not trust agent *A*.) One way to do this is to have the compiler *prove* that the target code correctly corresponds to the source code.⁴ Now, as it turns out, in the type theory of SML, not only can such a proof be written out formally, but in fact it can be written in a typed language with the property that any well-typed proof is guaranteed to be valid.

Proof-carrying code is thus an application of ideas from programming-language theory, in this case used for defining notions of safety that are useful for operating systems, and flexible enough to accommodate both high-level and low-level languages. With the growth of interest in highly distributed computing, web computing, and extensible kernels, it seems clear to us that ideas from programming languages are destined to become increasingly critical for robust and good-performing systems.

7 Acknowledgements

We thank Robert Harper, Brian Noble, Daniel Jackson, Edo Biagioni, Greg Morrisett, Scott Draves, Chris Colby, Martin Abadi and Dave Detlefs for reading previous versions of this paper and for suggesting many improvements. We also thank Charles Garrett, Brian Bershad, Wilson Hsieh for suggesting many improvements to the methodology for the Modula-3 performance measurements. Finally, we thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Jay Lepreau, who also suggested the PCC name.

References

- [1] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (Dec. 1995), pp. 267–284.

⁴This is essentially the same as having a compiler translate the types as well as the code, so that the target program will have types that can be checked. In fact, this approach to compiling is taken by the SML/TIL compiler [22].

- [2] BOYER, R. S., AND YU, Y. Automated proofs of object code for a widely used microprocessor. *J. ACM* 43, 1 (Jan. 1996), 166–192.
- [3] CLUTTERBUCK, D., AND CARRÉ, B. The verification of low-level code. *IEEE Software Engineering Journal* 3, 3 (May 1988), 97–111.
- [4] CONSTABLE, R. L., ALLEN, S. F., BROMLEY, H. M., CLEAVELAND, W. R., CREMER, J. F., HARPER, R. W., HOWE, D. J., KNOBLOCK, T. B., MENDLER, N. P., PANANGADEN, P., SASAKI, J. T., AND SMITH, S. F. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [5] DIJKSTRA, E. W. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM* 18 (1975), 453–457.
- [6] FLOYD, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. American Mathematical Society, 1967, pp. 19–32.
- [7] HARPER, R., HONSELL, F., AND PLOTKIN, G. A framework for defining logics. *Journal of the Association for Computing Machinery* 40, 1 (Jan. 1993), 143–184.
- [8] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM* 12 (1969), 567–580.
- [9] HSIEH, W. C., FIUCZYNSKI, M. E., GARRETT, C., SAVAGE, S., BECKER, D., AND BERSHAD, B. N. Language support for extensible operating systems. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 127–133.
- [10] LEE, P., AND LEONE, M. Optimizing ML with run-time code generation. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 137–148.
- [11] MARTIN-LÖF, P. A theory of types. Technical Report 71–3, Department of Mathematics, University of Stockholm, 1971.
- [12] MCCANNE, S. The Berkeley Packet Filter man page. BPF distribution available at <ftp://ftp.ee.lbl.gov>, May 1991.
- [13] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference* (Jan. 1993), USENIX Association, pp. 259–269.
- [14] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
- [15] MOGUL, J. C., RASHID, R. F., AND ACCETTA, M. J. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles* (Nov. 1987), ACM Press, pp. 39–51. An updated version is available as DEC WRL Research Report 87/2.
- [16] NECULA, G. C., AND LEE, P. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Sept. 1996. Also appeared as FOX memorandum CMU-CS-FOX-96-03.
- [17] NELSON, G. *Systems Programming with MODULA-3*. Prentice-Hall, 1991.
- [18] SIRER, E. G., SAVAGE, S., PARDYAK, P., DE-FOUW, G. P., AND BERSHAD, B. N. Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 134–140.
- [19] SITES, R. L. *Alpha Architecture Reference Manual*. Digital Press, 1992.
- [20] SUN MICROSYSTEMS. The Java language specification. Available as <ftp://ftp.javasoft.com/docs/javaspec.ps.zip>, 1995.
- [21] SUN MICROSYSTEMS. The Java Virtual Machine specification. Available as <ftp://ftp.javasoft.com/docs/vmspec.ps.zip>, 1995.
- [22] TARDITI, D., MORRISSETT, J. G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 181–192.
- [23] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles* (Dec. 1993), ACM, pp. 203–216.
- [24] WALLACH, D. A., ENGLER, D., AND KAASHOEK, M. F. ASHs: Application-specific handlers for high-performance messaging. In *ACM SIGCOMM'96* (Oct. 1996), vol. 26, ACM.

An implementation of the Hamlyn sender-managed interface architecture

Greg Buzzard, David Jacobson, Milon Mackey,
Scott Marovich, and John Wilkes

Computer Systems Laboratory,
Hewlett-Packard Laboratories, Palo Alto, CA

As the latency and bandwidth of multicomputer interconnection fabrics improve, there is a growing need for an interface between them and host processors that does not hide these gains behind software overhead. The Hamlyn interface architecture does this. It uses sender-based memory management to eliminate receiver buffer overruns, provides applications with direct hardware access to minimize latency, supports adaptive routing networks to allow higher throughput, and offers full protection between applications so that it can be used in a general-purpose computing environment. To test these claims we built a prototype Hamlyn interface for a Myrinet network connected to a standard HP workstation and report here on its design and performance. Our interface delivers an application-to-application round trip time of 28 μ s for short messages and a one way time of 17.4 μ s + 32.6ns/byte (30.7MB/s) for longer ones, while requiring fewer CPU cycles than an aggressive implementation of Active Messages on the CM-5.

1 Introduction

Processors are rapidly getting faster, and message-passing multicomputer interconnections are doing the same, thanks to recent developments in Gb/s links and low-latency packet switches. But the cost of passing messages between applications also includes the overhead of crossing interfaces between the operating system (OS), a device driver, and the hardware, which can be orders of magnitude more than the cost of moving a message's bits across the wires.

Hamlyn is an architecture for processor-interconnection interfaces that addresses this difficulty. It achieves both low latency and high bandwidth, isolates applications from each other's mistakes, and supplies a rich set of message-delivery semantics. It does so by exploiting several techniques:

- *Sender-based memory management.* Senders, not receivers, choose the destination memory address at which messages are deposited. This means that messages are sent only when the sender knows that there is memory space for them, eliminating buffer overrun and retransmission under heavy loads.
- *Direct application access to interface hardware.* Send and receive operations require no OS intervention, yielding very low latencies.
- *Zero-copy protocols.* Data are transferred directly between application memory and the network with no memory-to-memory copying or page remapping.
- *Automatic message reassembly.* The interface allows out-of-order packet delivery in order to support adaptive routing networks, which have greater throughput and fault tolerance [Davis92].

- *Data movement and message arrival notification are separate.* Data can be moved without interrupting the remote host if desired, which provides greater application control and lower overheads [Thekkath94].

The original Hamlyn design [Wilkes92, Wilkes95], which incorporated most of these features, was intended to support a packet-based, fault tolerant, adaptive routing network for a large-scale, MIMD multicomputer, derived from the Mayfly project [Davis92]. This paper extends the original Hamlyn work by describing:

- performance data from a working prototype;
- improved methods of message arrival notification;
- a more powerful packet counting scheme that supports generalized group-receive semantics;
- layered protocols that provide in-order message streams and application buffer management.

We describe the Hamlyn architecture and our application interface library, presents performance measurements of our prototype, discusses related work, and then summarizes what we have learned.

2 The Hamlyn architecture

Hamlyn was intended to support scalable, concurrent, fault-tolerant applications, running on a MIMD multicomputer or a closely-coupled computer cluster. Such applications often require high-bandwidth bulk data transfers, low-latency control messages (a few microseconds per round-trip), and multiple, independent protection domains provided concurrently on each processor.

For low latency, Hamlyn gives applications direct access to the interface hardware for sending messages with no OS intervention. It provides a fast, low-cost, message-arrival notification mechanism that does not require interrupts or system calls in order to receive messages. (Interrupts may be used as an optional alternative, in which case Hamlyn tries to coalesce them instead of delivering one for each packet.)

For high bandwidth, Hamlyn includes a scatter-gather direct memory access (DMA) capability that frees the host processor for other operations during long transfers. Applications can use it directly: it does not require OS intervention for each use. This allows Hamlyn to avoid all memory-to-memory copying in the host.

For security in a multiple-user environment, Hamlyn prevents mutually suspicious applications from sending, reading, or overwriting each other's data even though they use the interface hardware directly. This avoids the need to statically partition a multicomputer, or having to use gang-scheduling and interconnection fabric draining for inter-application protection, as on the CM-5.

Hamlyn deliberately exploits several features of the short-distance interconnection networks commonly used in modern, multicomputer systems:

- *Very low transient error rates.* Dedicated, enclosed, multicomputer interconnection fabrics are better thought of as extended backplanes than as unreliable networks—they rarely lose or corrupt packets. Hard failures can occur, but transient errors are so infrequent (perhaps one every few months) that it is reasonable to handle them using high-level, application-program mechanism, such as aborting and restarting a transaction [Saltzer84].

This means that automatic retransmission by a lower level of a protocol stack is unnecessary, improving performance; that packet reception need not be acknowledged, eliminating a potential cause of deadlock; and that transmission buffers can be released as soon as their contents have been sent, simplifying buffer management.

- *Hardware flow control in the interconnection fabric.* This avoids packet loss by applying back pressure to senders when resources fill up.
- *Small packet sizes.* These permit simpler, faster switches and better throughput guarantees, at the cost of requiring message segmentation and reassembly in the interface.
- *A physically secure network.* In such networks, messages need not be encrypted to protect them against eavesdropping.

Hamlyn was designed with a RISC-like philosophy: to make common cases fast and less common ones possible.

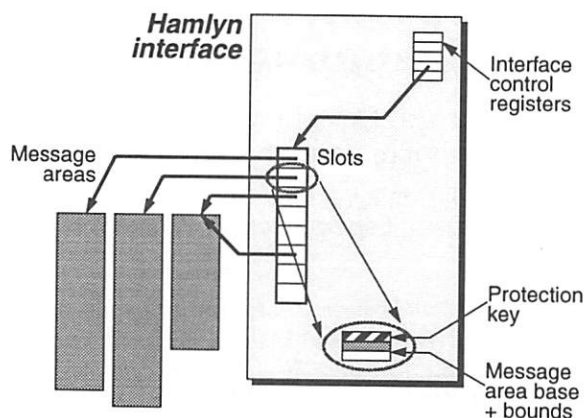


Figure 1: message areas and slots.

An explicit goal was that Hamlyn should be simple enough to be implementable using hardware state machines, since programmable controllers are often slow.

The next subsections describe the features of the Hamlyn design that allow these goals to be met.

2.1 Sender-based memory management

The first—and perhaps most important—feature is *sender-based memory management*, which is a technique to avoid software-induced packet loss. Packet loss is a serious problem in low-latency data communication systems because coping with it usually means retaining transmission buffers, acknowledging packet reception in order to release buffers when they are no longer needed, and using a low-level time-out mechanism to trigger retransmission. Moreover, the problem usually occurs under heavy loads, when retransmission will only make it worse, so even a low rate of packet loss can produce a much higher rate of message loss.

There are two main causes of packet loss: interconnection network problems, such as damaged or lost packets, and receiver buffer overrun. Our design assumptions legislate away the former, and we use sender-based memory management to prevent the latter.

The basic idea is to determine a message's final destination in a receiving host's memory before sending it, so that receiver buffer overrun is impossible. The receiving network interface places incoming packets directly into their final resting place instead of leaving them on an interface card or temporarily copying them elsewhere in the receiving host's memory.

2.2 Slots: naming and protecting message areas

Data are sent to and from *message areas*, which are contiguous regions of an application's virtual-memory address space, protected by OS mechanisms in the usual way (Figure 1). Message areas are *wired down* (pinned

into memory) while they remain allocated. This is a deliberate design decision that trades greater physical memory use for lower latency and greatly simplified interface design.

Message areas are referred to by *slots*, which are in turn indexed by small integer *slot numbers*. A slot contains base and bound registers for the message area to which it refers (several slots may refer to the same message area), and a protection key that must also be held by applications wishing to send messages to it. A slot is implemented by a data structure that can only be modified by the OS; this data structure lives in the network interface hardware in our prototype.

Memory addresses in Hamlyn messages are represented by <slot-number, offset> tuples; this indirection allows senders to be isolated from the details of virtual and physical memory addressing at receivers. Since packets can potentially arrive out of order, each one needs to be self-describing. This is accomplished by having the Hamlyn interface add a header to each packet that it sends out, as shown in Table 1.

Table 1: packet header format (slightly simplified). Each line represents 32 bits.

Destination host ID	
Slot number	Metadata index
Protection key (64 bits)	
Packet offset	
Packet length	
Delta (used in packet counting)	
Metadata length	Flags
(user data follows here ...)	

When a packet arrives at a receiving interface (Figure 2), the interface locates the named slot, adds the base address of the target message area to the offset specified in the header, and then moves the packet's data to that address using DMA, after checking that the packet will fit into the message area. When the last packet of a message arrives the interface will also notify the receiving application if desired (see section 2.6).

To ensure that data cannot be written into a buffer without permission, the receiving interface compares the protection key in the packet header to the one in the slot. Only if the keys match is writing allowed. Hamlyn protection keys are large (64 bits) and sparsely allocated to provide good inter-application protection.

Application software can often be simplified if a message carries a small amount of out-of-band *metadata* to be deposited in a separate buffer, so our prototype Hamlyn interface allows up to 60 bytes of it in the first packet sent. Secondary base and bound registers and a protection key

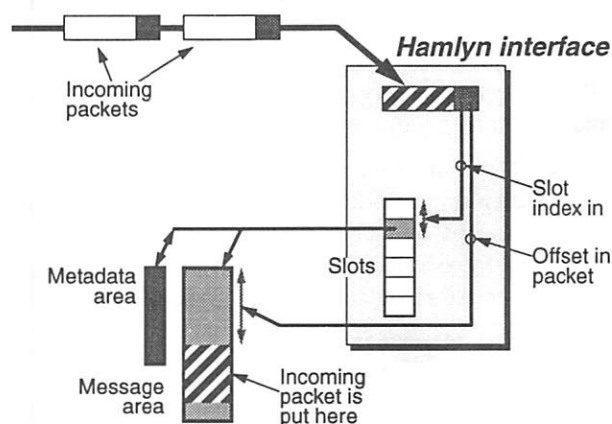


Figure 2: processing an incoming packet.

may be used to prevent unauthorized overwrites of metadata at the receiver (see section 2.7).

2.3 Sending termini

Hamlyn gives each sending application a private hardware *send terminus*, implemented as a set of control registers and a FIFO work queue in the interface card. These are mapped into the application's virtual-memory address space and protected by OS mechanisms in the usual way. In our prototype, each work queue holds up to 63 entries, so that applications can quickly post several messages without blocking. When the interface sends a message, it writes a sequence number in a prearranged, per-terminus, host memory word; this number, modulo the work queue's size, identifies the corresponding entry, thereby telling the application that the entry and any buffer memory associated with its message can be reused.

Short messages are pushed from a host processor to the terminus queue using ordinary STORE instructions. We call this *direct I/O*. To send a message, the application writes a transmission-request block—basically a packet header—into the send terminus' work queue, followed by any metadata, then the data. It then notifies the hardware of the new entry and proceeds to other work; no system call or interrupt occurs when sending a message.

Long messages are pulled from host memory by the interface using asynchronous DMA. Our prototype prevents applications from accessing data belonging to other applications in the following way. Each send terminus has 8 base and bounds registers, settable only by the OS, that are used to identify special *send buffer areas* that are named by small integer *send buffer tags*. A single message can contain parts from one or more of these areas; the Hamlyn DMA engine interface gathers them up on the fly as the message is sent out.¹ To send a long

¹ Our original proposal [Wilkes92] used slots instead of special send message areas for this purpose.

message, the application writes a transmission-request block to the terminus' work queue that consists of the header followed by a sequence of <send_buffer_tag, offset, length> tuples describing the location of metadata and data. It then notifies the interface of the new entry.

The send terminus automatically segments messages larger than a single packet, replicating the header in each packet—except for the offset field, which is adjusted automatically to reflect the address of the new packet at the receiver. All metadata are put into the first packet.

The Hamlyn interface interleaves packets from all the send termini with non-empty work queues, providing approximately equal bandwidth to competing processes sending large amounts of data and, in the absence of network delays, bounding the time any message waits in a work queue. This scheme could be embellished with priorities, although our prototype didn't include them.

2.4 The metadata area

Message areas are intended to receive most incoming data, but we needed three other storage structures for each arriving message:

- a place in which to deposit metadata;
- packet counters for the message-arrival notification mechanism (see section 2.5);
- information for a finer-grained, per-message protection scheme (see section 2.7);

There might be hundreds of application processes, with hundreds of interleaved, concurrently arriving messages per process, so we might need tens of thousands of these structures—too many to store on the interface card.² We chose one mechanism to solve all of these problems: a 128-byte *metadata entry* is provided for each expected message (Table 2). These entries are arranged in a vector, called a *metadata area*, whose base and bound are stored in a slot data structure. It lives in a receiving application's virtual-memory address space. Each packet header specifies the index of a metadata entry associated with its destination slot, which may be thought of as a message identifier. A metadata entry may be reused as soon as all packets of a message referring to it have been assembled by the interface and processed by the receiving application. A slot must therefore have enough metadata entries associated with it to accommodate the largest number of messages that might arrive concurrently.

2.5 Packet counting

In some interconnection fabrics, packets of a segmented message may arrive at a receiving interface out of order, so a mechanism is needed to determine when all the

²The Myricom LANai 2.3 network controller IC that we first planned to use only supported 128K bytes of memory for all of the control program, slot and terminus data structures, and packet buffers.

Table 2: a metadata entry.

<i>User metadata</i> (60 bytes)
Packet accumulator
Paranoid-mode protection key
Paranoid-mode message area low limit
Paranoid-mode message area high limit
Hamlyn library receive class pointer

packets of a message have arrived and the receiving application can be notified. Hamlyn does this by maintaining a 32-bit *packet accumulator* in the metadata entry for each expected message. The value of the accumulator starts out at zero and returns to this when all the packets have arrived [Jacobson95].

Each send terminus has a 32-bit *packet counter*, which is initialized to a value (Y) specified by the sender in the delta field of a transmission-request block. For each packet except the last in a message, the send terminus sets a 32-bit *delta* field in the packet header to 1 and decrements its packet counter using 2's complement arithmetic. For the last packet it sets the delta field to the final value of the packet counter. It is easily seen that the sum of all delta fields in a transmitted message's packet headers equals Y , modulo 2^{32} .

In receiving interfaces, the delta field in a packet is added to its associated packet accumulator as the packet arrives. When the accumulator reaches zero again, the entire message has arrived and a receiving application can be notified (see section 2.6). If a packet delta field is zero, notification occurs without consulting the packet accumulator. This is used as an optimization for single-packet messages.

This deceptively simple mechanism provides two important capabilities:

- **Single message receive.** Out-of-order packet arrival is handled as described above. To summarize: if a message has only one packet, then its sender sets the packet header's delta field to 0 and notification occurs immediately upon arrival. If a message has several packets, then the sum of the packet headers' delta fields is 2^{32} . Since the receiving packet accumulator is also counting modulo 2^{32} , its value will return to 0 exactly when all packets have arrived.
- **Group receive.** A single notification can be generated when a set of messages from a known group of senders has arrived (e.g., for a distributed barrier operation). In this case, the i th sender is given an initial value Y_i such that the sum over all participating senders of $Y_i = 2^{32}$. This sum reaches 2^{32} and wraps around to zero exactly when all packets in the message set have arrived. Scatter-

gather I/O is a special case of this in which all messages come from the same sender.

Moreover, a process receiving a packet counter value Y_i can delegate work to (say) two other processes, j and k , as long as their initial packet counter values, Y_j and Y_k , sum to Y_i . The identity of a group's senders need never be known by a receiver. The Hamlyn library (section 3) provides a means for dividing buffer space and counter values among the delegates of the group.

Individual, multiple-packet messages could be handled using a simpler mechanism in which each packet header's delta field carries the total packet count, and a packet accumulator is reset by the first packet to arrive; however this would require the sending interface to calculate the total number of packets before sending any. Our scheme allows group reception and scatter-gather I/O while a receiver remains oblivious to the number of packets sent, and requires no extra memory.

2.6 Message arrival notification

A Hamlyn interface indicates that all packets of a message have arrived by appending an entry to a circular *notification queue* in main memory. There is one such queue for each receiving application. The interface then generates an interrupt if requested, the receiving process is asleep, and the processor has no pending interrupt requests from the interface for other processes. All this reduces interrupt overheads to minimum.

Notification queues are identified by *notification queue control blocks* (NQCBs) in the interface card's SRAM, which are in turn referred to by slot data structures (Figures 3 and 4). Each NQCB has a cursor that points to the tail of its notification queue; when all of a message's packets have arrived, an entry (Table 3) is written at the location indicated by the cursor and the cursor is advanced. When it advances beyond the queue's storage area, the cursor is reset and a wrap flag in the NQCB is toggled—that is, the notification queue is treated as a circular list.

Table 3: notification queue entry.

Wrap flag	Slot index	Notification index
	Metadata entry pointer	
	(padding for alignment)	

A receiver application wishing to busy-wait for a message maintains its own cursor and wrap flag and polls the next available notification queue entry until the wrap flags match. (The cache-coherent I/O of our prototype's workstation ensures that this busy-waiting causes no I/O bus activity until the entry is rewritten.) Because the notification queue is read-only by the application and

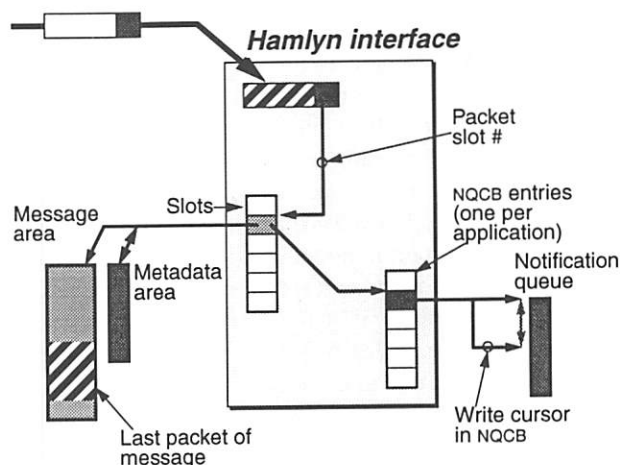


Figure 3: notification queues and message notification.

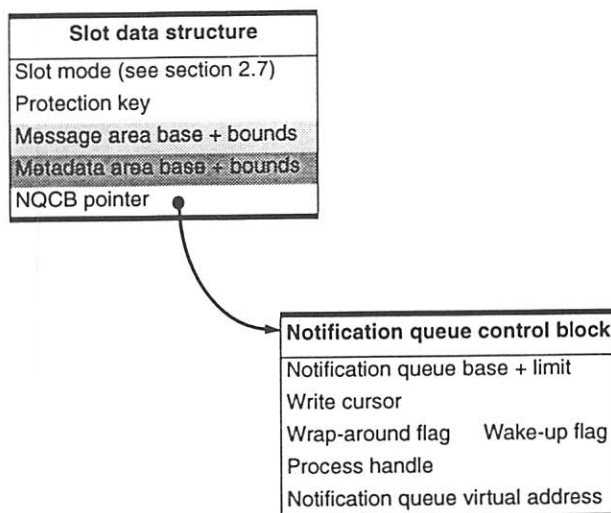


Figure 4: slot and notification queue control block contents. Several slots may point to the same NQCB.

write-only by the interface, no other locking or synchronization is necessary.

This is the only part of the Hamlyn design in which buffer overrun might occur: a faulty or malicious sender could transmit messages faster than a receiver can consume them, overwriting an older notification queue entry. This can be prevented by ensuring that the queue's size exceeds the number of messages arriving in the worst case. Section 2.7 describes a mechanism that can enforce an upper bound on this count; section 2.8 describes a discipline-based solution to bounding it.

2.7 Special modes of operation

The mechanism described so far accommodates cooperating processes in a single application, but we wanted Hamlyn to provide increased robustness, privacy,

and security for client-server systems of mutually suspicious processes: specifically, several senders sharing a common receiver slot. To this end, slots can be put into two special modes of operation called `paranoid` and `paranoid_one_shot`. Both modes use extra fields in a message's metadata entry: a secondary base and bounds register, and a protection key that supersedes the slot data structure's key. When a packet arrives in a slot operating in one of these special modes, the packet header's base and bound fields are compared first to the slot's message area (primary) base and bounds, then to the metadata (secondary) base and bounds, while the packet header's key field is compared to the secondary protection key. All tests must pass before the packet is accepted. Additionally in `paranoid_one_shot` mode, no further use of the metadata entry is allowed without application software intervention after the first message for the entry has been received, preventing subsequent messages from being received before the first one is processed.

These modes provide several advantages:

- Senders' data can be confined to a small part of a message area, limiting the damage that a faulty or malicious client can cause.
- The `paranoid_one_shot` mode prevents a faulty or malicious client from overrunning a notification queue as long as the queue's size exceeds the number of metadata entries used. This mode also ensures that a packet's data cannot be overwritten after message arrival, so the data need not be copied elsewhere for safe-keeping.
- Every sender can have its own protection key, allowing revocation of one sender's access rights without affecting others. For example, if a node appears to have failed, all of its senders' keys can be revoked, potentially allowing message and metadata areas to be immediately reassigned.
- Since metadata areas are allocated in main memory, applications can change the secondary base, bounds, and key registers for their own metadata areas without OS intervention.

These modes entail more complicated interface logic and extra tests during packet arrival which introduce a small amount of extra latency. (There need not be more host memory accesses, since a metadata entry's packet accumulator must be updated anyway.)

A third special mode of operation, called `fast` mode, represents a special-case optimization for single-packet messages: if an arriving packet header's metadata index is all 1's, then the packet header is used to carry exactly one word of metadata, which is written in a notification queue entry in place of a metadata entry pointer. This lets a single-packet message carry a small amount of metadata

with minimal overhead. In retrospect, it may have been a premature optimization.

2.8 Flow control and deadlock prevention

An important consideration in Hamlyn's design was to ensure that, in the absence of a failing sender, receiver, or interconnection fabric, data is never lost and communication never deadlocks. Deadlock is a treacherous issue. If Hamlyn employed low-level acknowledgment of each packet's transmission, and communications were ever blocked because the fabric cannot accept more packets, then acknowledgments could be blocked as well, potentially causing deadlock.

We avoid deadlock by not depending upon hardware packet acknowledgments and by arranging that the only hardware moderating the flow of incoming packets is the host's I/O bus. Incoming transfers are given priority over outgoing transfers for access to the I/O bus, so that the maximum time an incoming packet stalls at the interface is the time for one packet to traverse the I/O bus plus a small amount of overhead in the interface.

When power is first applied to a Hamlyn interface card, it enters a state in which it simply accepts and discards arriving packets. It does the same if the host fails to reset a periodic handshake timer. The Myricom switch used in our prototype detects powered-down interfaces and discards packets destined for them. It uses a round-robin service discipline for incoming ports to avoid starvation of incoming packets. If arriving packets are delayed while waiting for the Hamlyn interface to process them, or if there is conflicting traffic at a switch port, the switch and interface hardware generate link-level back-pressure, halting the sender. Eventually the sending application will block when its terminus' work queue fills up.

To streamline higher-level protocols, we exploit the fact that the interconnection fabric never loses packets in the absence of failures, which is a reasonable design decision for a small-area, multicomputer interconnection, although less so for a wide-area network. Sender management of memory automatically imposes a higher level of flow control for buffer space and metadata management, so a sending application blocks when no resources are available to service a request at the receiver.

A final concern is to prevent notification queue overrun. This can be accomplished by ensuring that the gap between the number of messages processed at the receiver and the number that can have been transmitted by the senders is always smaller than the number of metadata entries available. This limit can be enforced in `paranoid_one_shot` mode.

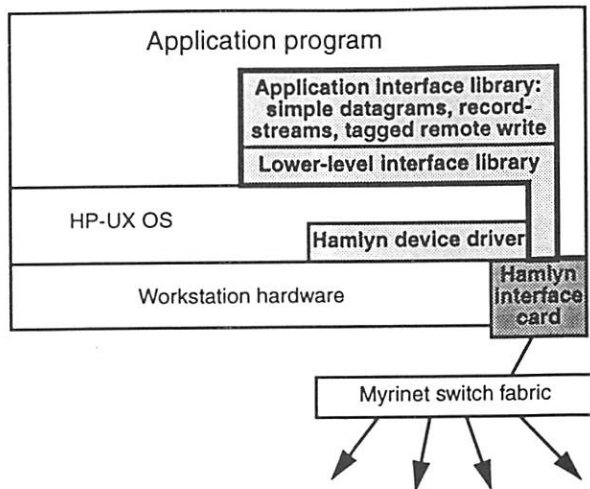


Figure 5: schematic view of Hamlyn hardware and software. The shaded portions were added to support Hamlyn.

2.9 Architectural costs

Hamlyn consciously makes some design choices that impose additional costs compared to more traditional approaches:

1. Message, metadata, and notification queue storage areas must be wired down.
2. Applications are responsible for buffer management, including reclamation after sends and receives.

In return for these costs—which we think are relatively small, and not unlike those imposed by other high-performance network designs—Hamlyn provides fully-protected, direct application access to a network, automatic message segmentation and assembly, group reception notification, rejection of messages from failing or malicious processors, and both direct I/O and DMA sends.

3 The Hamlyn interface library

In order to make the Hamlyn architecture easier to use, we built a two-level application interface library. The upper layer provides a set of convenient programming abstractions and hides the details of buffer memory management. The lower layer provides a simple, efficient procedural interface to our communication hardware. The library was designed to provide a convenient infrastructure for popular middleware, such as MPI [Corbett95], Active Messages [vonEicken92], and Oracle's distributed lock manager. We describe these layers from the bottom up (Figure 5).

3.1 OS interface

The Hamlyn interface card is managed by a device driver module in the host operating system. OS modifications to

support Hamlyn in UNIX systems³ are largely confined to this driver, which provides all interface management services requiring OS mediation, such as creating slots and termini, installing slot protection keys, wiring and unwiring memory, and arranging to suspend or resume an application pending a message's arrival.

All other interface management services reside in unprivileged library code, linked in with application programs.

3.2 Low-level library procedures

The Hamlyn library includes a procedural interface to the network interface hardware and hardware-manipulated data structures. It uses a data structure called a *ticket*, which contains a message's destination, slot number, metadata index, protection key, data buffer base and bound, and some flags. Tickets are location-independent in that they may be exported in a message and used later to reply from a remote Hamlyn interface.

The library's lower layer has two main functions: `h_send_msg` and `h_rcv`. The former accepts a ticket, a data buffer's address and length, and a metadata buffer's address and length. Buffer addresses are converted to offsets in sender message areas. A small message is written to the interface using direct I/O, while a DMA request is built for a large message. The function returns a handle that can later be used to decide when to release a buffer.

The `h_rcv` function checks whether a message has arrived and, if so, it returns the address of the corresponding notification queue entry. A variant, `h_rcv_block`, accepts a time-out argument and waits until either a message arrives or the specified interval expires.

3.3 Higher-level protocols

The Hamlyn library's upper layer supports a set of protocols with varying semantics to send and receive data. Three protocols, embodying most of its key ideas, are described below.

This layer was written in C++ because the language lets communication end points be represented as objects, keeping an application's name space clean and letting each protocol use the same operation names (*e.g.*, send, receive). We were inspired by the work of Stepanov and Lee on the C++ Standard Template Library [Stepanov95] in which aggressive inlining and optimization are combined to achieve highly efficient object code.

The library creates a Hamlyn manager (an instance of the `hamlyn_manager` class) for each send terminus. This class is responsible for managing the device driver interface and notification queues, and for allocating buffer memory.

³ UNIX is a registered trademark of X/Open Company, Limited.

Applications create an end point for message reception by instantiating a receiver class, which contains one or more metadata areas and buffers, and a queue of received messages. It also provides a C++ virtual procedure (process_arrival) that is called by the Hamlyn manager to record a new message's arrival.

When receive is applied to an end point, a message is returned from the arrival queue in the receiver class instance if possible. Otherwise—if the queue is empty—the receiver instance calls a poll_nowait routine in the hamlyn_manager. The latter gets the next notification queue entry if there is one, follows its pointers to the metadata area and receiver end point, and then calls process_arrival there, passing it the address of the metadata. Control then returns to the receiver end point originally called by the application, which looks again for a new message. (It may not have received one if the notification queue was empty or the queue entry just processed represented a message for a different receiver instance.) This process continues until the original request is satisfied, a time interval expires, or the receiver end point chooses to block instead of busy-wait.

Although the Hamlyn architecture and the library's lower level are thread-safe, the upper level is not. Making it so remains a research topic.

All of the connection-oriented receiver classes support a make_seed call, which returns a seed object, containing tickets for preallocated buffers, metadata, and other information. A seed can be sent to a remote node in order to create an instance of the corresponding end point sender class. (It's so named by analogy with the similar purpose seeds serve in plants.)

The Hamlyn library uses these techniques to support the following protocols:

Simple datagram protocol. This provides access to raw Hamlyn hardware semantics. The send call is a wrapper for the h_send_msg routine. It creates a small amount of metadata that tells the receiver the offset and length of the transmitted data and can include a reply ticket. There is a separate receiver instance for each metadata entry, and each instance is either "ready" or "not ready": there is no queue.

Stream protocol. This provides a one-way, one-to-one, in-order connection from a sender to a receiver. The sender class supports send and flush. Large buffers are sent as-is, and small ones may be coalesced by copying. Calling flush forces transmission of all previously-posted data. The receiver class supports receive and release; the former returns a <start-pointer, length> pair describing its result; no copying occurs. It automatically allocates more buffers if needed. The release procedure frees all records up to and including that identified by its argument.

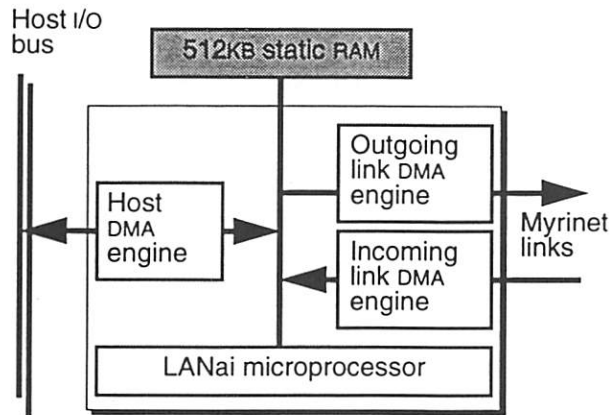


Figure 6: schematic view of the Myricom network interface.

Senders block if they ever get so far ahead of the receiver that they run out of tickets (which provide permission to write to a metadata entry).

Tagged Remote Write. In this protocol, a send call specifies a ticket, a source buffer's address and length, the destination buffer offset, and an integer tag. Tags are enqueued in the receiver and can be retrieved by calling get_tag or get_specific_tag. This protocol uses fast mode, so messages must fit in a single packet. In-order delivery is not guaranteed.

4 Performance evaluation

In order to evaluate the Hamlyn architecture, we collaborated with the University of California at Berkeley and Myricom, Inc., to build a prototype interface card for a Myrinet network [Boden95, Buzzard95].

The Myrinet switch we used is a non-blocking, 8x8 crossbar, which uses wormhole routing. It provides 80MB/s of bandwidth per port in each direction with about 0.5μs of latency. We used Myricom's LANai Version 4.0 network controller chip with 512KB of on-card static RAM, microcoded to implement the Hamlyn design. The LANai has a 32-bit CPU and three DMA units (Figure 6): incoming from the switch, outgoing to the switch, and to/from host memory. The host DMA engine is the only mechanism available to the LANai controller to access the host's main memory.

Our host computers were early-production HP 9000 Series 770 (J200) PA-RISC workstations with 100MHz CPUs, running Version 10.00 of the HP-UX operating system. The interface cards plugged into the workstations' graphics I/O bus, which operates at the same frequency as the LANai CPU (40 MHz). The bus and its processor interface can support incoming DMA at 106MB/s, but outgoing transfers

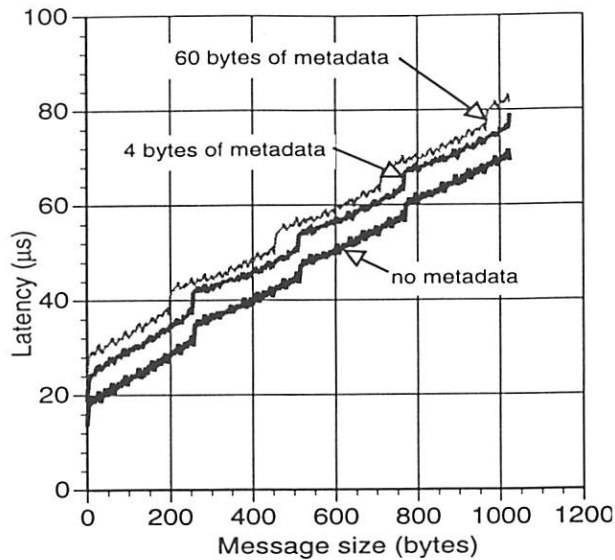


Figure 7: latency vs. message size for 256-byte packets.

are limited to 32MB/s because of reduced opportunities for pipelining requests through the bus interface chip.

The workstations have a cache-coherent I/O architecture, which obviates the need for software to flush or purge data cache lines in DMA buffers. It also lets applications busy-wait for Hamlyn DMA completion without consuming I/O bus bandwidth. DMA buffers' I/O bus addresses are mapped to physical memory addresses by the workstations' memory-to-I/O bus interface hardware, so that our card can access a multiple-page buffer in a contiguous I/O bus address range—a considerable simplification.

Unless otherwise stated, our performance measurements were taken by sending a message from one application-level process to a second remote one, which returned the message to the sender. The round-trip times we measured were divided by two to get one-way data. Where we report single measurement numbers, we generated them by timing at least 10 000 messages. There was less than 1% variation between independent runs of our test suite.

The performance data reported here apply only to our test systems and do not necessarily represent products currently in production.

4.1 Short single-packet messages

The lowest latency is obtained using our interface's fast mode of operation (see section 2.7). We measured two cases using 16-byte payloads. In one, application code wrote data to the interface using direct I/O without the Hamlyn library, while the other case used the library's Tagged Remote Write protocol. The first case took 12.7μs one way (25.4μs round-trip), and the second case added

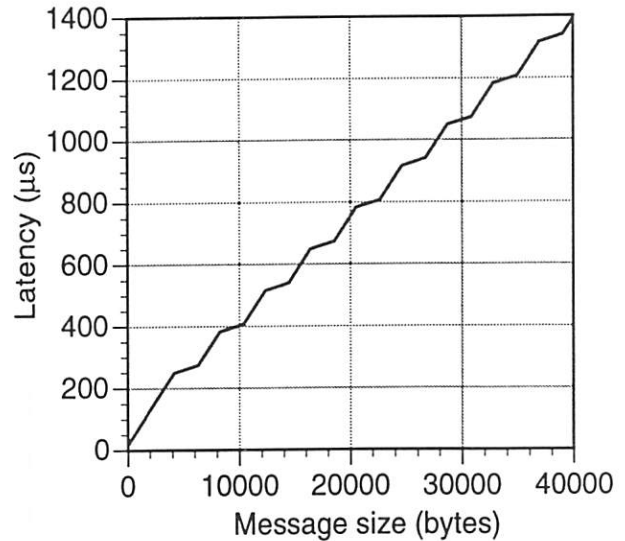


Figure 8: latency vs. message size for 4KB packets.

1.4μs of overhead, yielding 14.1μs in all (28.2μs round-trip). Table 4 shows where the time goes.

Table 4: one-way short-message transfer time (μs).

	raw interface	tagged-remote-write protocol
DMA	3.3	3.3
LANai	6.7	6.7
Switch	0.5	0.5
Host I/O writes	1.4	1.4
Host protocol software	0.8	2.2
<i>Total</i>	12.7	14.1

Notice that the 1.4μs due to the host I/O writes represents only 8 STORE instructions: these are slow because of the cost of traversing the I/O bus. By contrast the host protocol software costs represent tens or hundreds of instructions.

4.2 Latency for normal messages

Figure 7 shows the one-way latency as a function of message size with 0, 4, and 60 bytes of metadata and 256-byte packets. Several effects are visible:

- A baseline cost of about 17.4μs due to host software overhead, LANai control program overhead, and the cost of writing a notification queue entry.
- Increases in latency at message sizes that are multiples of 256 bytes. Our LANai code takes about 12.5μs more to receive a packet of this size than to send one, so the overall time increases at each packet boundary. The first step is larger than the others because the code changes from never updating the packet accumulator to updating it twice, while subsequent packets cause one update each.

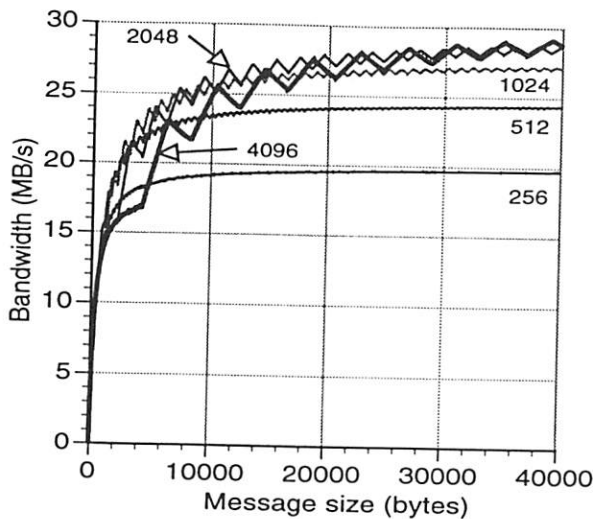


Figure 9: bandwidth vs. message size for packet sizes of 256, 512, 1024, 2048, and 4096 bytes and no metadata.

- A marginal transmission cost of 55.5ns/byte (18.0MB/s). This results because the three steps involved are handled serially for each packet: about 31.3ns/byte to move outgoing data to the interface using DMA, 12.5ns/byte to move it across the network at 80MB/s, and 11.7ns/byte for DMA into the destination host.
- Repeating fine structure with a 32-byte period due to power-of-2 I/O bus transaction sizes. For example, a 28-byte transfer requires three transactions (for 16, 8, and 4 bytes) while a 32-byte transfer is done in just one.
- An extra 7.7μs to sending metadata, including low-level library overhead to translate buffer addresses, then start incoming and outgoing DMA. Metadata bytes incur the same transfer cost as other data but are not counted in the x-axis of Figure 7, so sending more metadata sent shifts the lines to the left.

Figure 8 shows the one-way latency for 4KB packets. Here, the bottleneck is moving data from the sending host to the Hamlyn interface card across the I/O bus controller. The latency exhibits alternating costs of 55.5ns/byte and 11.6ns/byte: if the last, partial packet of a message is almost full, each additional byte waits for outgoing DMA, transmission, and incoming DMA, totalling 55.5ns/byte. But small, final packets arrive early enough that they need only wait for the previous packet's DMA into the host memory to finish, which sets the marginal cost of 11.6ns/byte.

If a receiving process is asleep, latency is dominated by interrupt service and process-context switching time. We observed⁴ 78μs for packets with no payload, which compares favorably with other recent reports [Jones96,

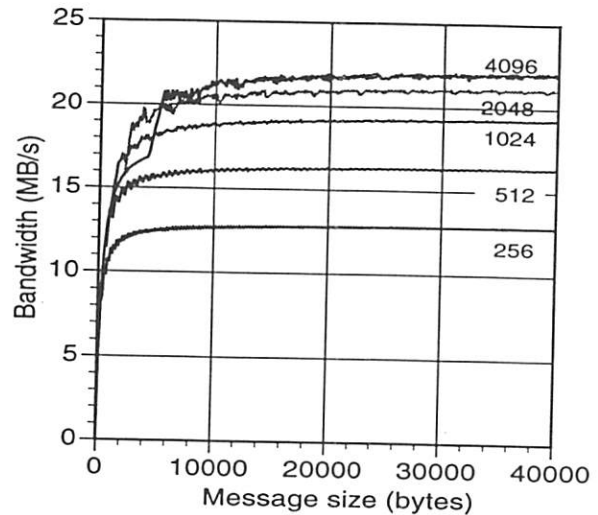


Figure 10: loopback bandwidth vs. packet size for packet sizes of 256, 512, 1024, 2048, and 4096 bytes.

Keeton95, vonEiken95]. (On the same machine, a context switch provoked by a semaphore takes 31μs.)

4.3 Bandwidth and packet size

Using 4KB packets, the bottleneck is the I/O bus interface. The observed slope of the latency function in Figure 7 is 30.7MB/s, from which we infer that the LANai control program achieves a 96% payload utilization of the 32MB/s outgoing DMA channel. The actual utilization is somewhat higher, since counter values are fetched and stored for each packet. (A potential optimization that we did not explore would be to cache some counters in the interface card.)

Our packet size can be altered by recompiling the LANai control program, so it is instructive to examine the latency achieved versus message size for various packet sizes. Figure 9 shows the overall one-way bandwidth as a function of message length for several packet sizes. The receiving interface takes 12.9μs per packet. For 256-byte packets, this limits bandwidth to 20MB/s, which agrees with our observations. Asymptotic performance increases with packet size, but 2KB packets outperform 4KB packets for smaller messages because there is more concurrency between outgoing DMA, transmission, and incoming DMA. The advantage of 4KB packets is slight and inconsistent up to message sizes of 40KB, the largest we measured.

Even though the interface does not pad packets to their maximum size, there are conflicting pressures on the choice of packet size:

- Choosing a packet size large enough that network transit time exceeds LANai packet-processing time

⁴ Our prototype had a defect, which was impractical to fix, that caused the actual times to vary widely. This was the minimum latency, which we are confident would be the latency in a corrected system.

keeps the LANai from being a bottleneck. The larger the packet size, the more LANai time remains for other tasks, including processing packets moving in the opposite direction.

- Long delays should be avoided. Transferring a 4KB packet occupies a link and a DMA channel for about 50 μ s, a long time to block another packet needing the same resources. This argues in favor of shorter packets.

Taking these into account, we recommend a packet size of 1KB for the particular combination of bandwidths and overheads measured for our prototype.

The effect of competing for LANai CPU cycles and I/O bus bandwidth can be seen when a message's source and destination are the same process on the same host, as shown in Figure 10. With only 4 bytes of data, the latency in this loopback test rises to 19.1 μ s because sending and receiving compete for the same LANai controller chip. The bandwidth figures tell a similar story: the asymptotic bandwidth of 28MB/s from Figure 9 drops to 22MB/s in Figure 10 because of I/O bus contention.

4.4 Projections for alternative hardware

The LANai performance is relatively low because it is a programmable controller. If we were to implement a Hamlyn interface using hardware state machines, we estimate that the one-way, application-to-application short-message transfer time would decrease to about 6 μ s, but the large-message bandwidth, which is limited by the I/O bus, would not change appreciably.

5 Related work

There has been a great deal of work in the field of interface design for high-speed interconnections, especially since the original Hamlyn design was written up. The history of these ideas is not entirely clear: several teams were inventing similar-sounding approaches at around the same time. It is neither fair to say that Hamlyn copied from them, nor that they copied from Hamlyn. (For the record, the earliest extant reference to Hamlyn is dated July 1992.) Although we think that Hamlyn's main contribution lies in its coupling of sender-based memory management to its protection scheme, we present a somewhat broader summary of what we consider to be the work most relevant to our finished design.

IBM's OS/360 provided variants of put and get file-system calls that avoided data copying by having the OS specify the location of the buffer to use, rather than the application [Clark66, Belady81]. Hamlyn uses a variant of this mechanism in its interface library.

5.1 Load/store interfaces

A STORE instruction can be thought of as a degenerate, sender-directed message—indeed, there is a large and active literature that views large-scale shared memory machines in this way, of which the Cray T3D, Convex Exemplar, KSR AllCache architecture, Alewife [Kranz93], Typhoon [Reinhardt94], and low-level SCI protocols [IEEE92] are representative examples. All require dedicated hardware support that is tightly integrated with the host processors.

Several groups have used the LOAD/STORE paradigm in less tightly coupled systems to provide an interface to cross-network communication. For example, the Alto remote memory reference protocol [Spector81] used network messages in this way; [Thekkath94] discusses the idea of separating data movement from notification in remote LOAD/STORE operations (Hamlyn also allows this); and SHRIMP [Blumrich94] provides low-latency remote-memory access using hardware support for automatic data replication, coupled to a virtual-memory protection scheme. Many of these schemes provide excellent performance for the particular operations that they support—specialization is a powerful tool for lowering latency—but sometimes at the expense of relatively high processor utilization. Most implicitly depend upon in-order packet delivery.

The hybrid deposit model [Osborne94] combines sender-based addressing with the execution of small programs on a remote node, using both local and remote data—a considerable generalization of the *remote fetch-and-op* proposed in [Wilkes92]. Implemented in software on top of a 155Mb/s ATM system, it achieved a round-trip time of 49 μ s without a switch and 60 μ s with one. Osborne credits [Subhlok93] with introducing the term “deposit model” for what we call sender-based memory management.

5.2 Copy avoidance

Several projects have used page-remapping and smart interface buffer allocation to accelerate processor-to-interface communication, including the *fbufs* work at the University of Arizona [Druschel93], the Medusa FDDI interface [Lumley92, Banks93] and the follow-on Afterburner project [Dalton93].

The Nectar system [Cooper90] allowed applications direct access to its communication interface memory in order to eliminate copies at the cost of all accesses being to memory in the I/O space. It achieved round-trip RPC latencies of 500 μ s across a 100Mb/s network.

ATM network interfaces can use virtual circuit identifiers (VCIs) to provide early demultiplexing of incoming data to user data buffers. One such use occurred in the Osiris project [Druschel94], which combined stream

demultiplexing using ATM VCIs into fbufs, some support for out-of-order delivery, and direct access to the network interface for a limited number of applications. Together, these achieved a round-trip latency of 154 μ s and a maximum throughput of about 41MB/s on a 622Mb/s ATM network.

CMU's Hardware-Assisted Remote Put (HARP) interface to the CreditNet ATM adapter card allows applications to send directly from their own buffers (akin to Hamlyn message areas), and to provide a set of buffers into which data for a virtual circuit is placed, but it does not appear to allow direct addressing of remote memory on a per-message basis [Mummert96]. We were unable to locate any published latency figures for this interface.

5.3 Cranium

Cranium [McKenzie94], like Hamlyn, was designed to provide a host interface to a packet-switching fabric that performed adaptive routing. Like Hamlyn, it has message areas that are used to send and receive data, although it appears that these are restricted to 2KB pages, and the expectation is that there is a single message per area, since there seems to be no provision for a message-offset field. Multiple packets use a sequence number to allow reassembly, rather than offsets; this simplifies the hardware, but it requires that all packets in a message be of the same size. (This means that Cranium could not handle variable-length metadata.) Receivers specify the identity of senders expected to write to a message area, and this is used for protection checks. (There is no discussion of how spoofing is prevented.) Cranium also provides queueing channels, which allow messages to be appended to the end of a message area. We thought about providing these for Hamlyn but eventually decided not to: (1) to force us to work through all of the details of pure, sender-based memory management; (2) to avoid introducing a *prima facie* source of receiver buffer overruns; and (3) because such messages have to be restricted to single packets. Cranium supports many of the goals of Hamlyn, but its designers made several decisions to reduce functionality in order to simplify the interface. It thus represents a different point in the design space.

5.4 Active Messages

Active Messages [vonEicken92, vonEicken94, Martin94] provide a set of arrival semantics for single-packet messages by including the address of a function to call in each one. The function is typically invoked in a restrictive environment on the interrupt stack, with no protection barriers around it. As a result, aggressive implementations of Active Messages are the standard performance target for this kind of work. The main difference is that Hamlyn provides security between applications; data placement is

controlled by the sender, rather than the receiver; and all protocol processing happens in a well-defined application context. "Although the restrictions and limitations of previous interfaces [to Active Message systems] made their implementations simple and efficient, the same restrictions and limitations prevent them from supporting the broader spectrum of applications now required" [Mainwaring95a].

[Karamcheti94] reported instruction counts (but no timings) for Active Messages on a CM-5 (CMAM), which are roughly comparable to ours although they were measured on a SPARC processor and ours are for PA-RISC. The CMAM finite-sequence, multiple-packet delivery protocol seems to provide functionality that approaches our simple datagram protocol: it does not support our group-receive operations, but, like ours, it does handle out-of-order packet delivery. [Karamcheti94] quotes 397 instructions to do a 16-word (64-byte) unidirectional send. A send using Hamlyn's tagged remote write consumes 260 processor cycles (fewer instructions), most of which are consumed when the processor stalls while writing to the I/O bus, and a receive consumes 120 cycles. [Wallach95] reports the lifting of one of the restrictions on Active Messages: that the handlers must not block.

We think that the idea of Active Messages is good, and we are gratified that some Hamlyn features are making their way into a revised proposal [Mainwaring95a], which supports protection, caching end point descriptions,⁵ as well as multiple send and receive areas per end point.

On the other hand, we think that a scheme requiring host processor intervention on every packet would **not** be such a good idea because the process-context switches would prove too expensive. Indeed, the current trend in processor design seems to be toward ever-larger amounts of machine state, which will make this more costly still. Hamlyn addresses this concern by automating message reassembly in the interface card.

U-Net [vonEicken95] embodies some of the same principles as Hamlyn, including direct user-level access to the interface in order to eliminate OS involvement whenever possible, and end points that can route incoming messages directly to application memory. Like Hamlyn, the prototype U-Net implementation is built by re-microcoding an existing interface card—a Fore Systems ATM interface. (By their definition, Hamlyn is using a "standard network interface"!) Since it is built on ATM, which is inherently unreliable, U-Net has to deal with lost packets. Its performance is slightly worse than Hamlyn's: [vonEicken95] reports Active Message round-trip times on top of U-Net of 79 μ s for 32 bytes of data or

⁵ [Wilkes92] suggested the same idea as a way to conserve interface-card memory.

less (41 μ s of which is due to the ATM switch; the equivalent Myrinet time is 1 μ s) and 135 μ s + 0.2 μ s/byte for bulk data transfers. (The equivalent Hamlyn numbers are probably the 28 μ s for a round-trip tagged remote write, and the one-way bulk data transfer cost of 17.4 μ s + 32.6ns/byte with 4KB packets.)

6 Conclusions

What did we learn from this exercise? First, the basic Hamlyn approach seems to have been validated: we can provide low-latency, high bandwidth, protected communication directly from multiple application programs, with little or no OS intervention. We also picked up a few other observations and lessons along the way.

6.1 Network demands

[Karamcheti94] argues that the underlying network should provide in-order delivery, deadlock freedom, and fault-tolerant packet transmission. We conclude instead that Hamlyn can synthesize in-order delivery cheaply, assuming a deadlock- and error-free network, giving interconnection designers freedom to optimize for performance, rather than high-level protocol support. [Davis92] argues that an adaptive-routing network can achieve roughly twice the throughput of a non-adaptive one.

6.2 Buffer management

Hamlyn does not copy outgoing messages, so applications must be coded to avoid reusing buffers until transmission is complete. To help with this, the Hamlyn library provides a function that determines whether a message buffer can be reclaimed.

Metadata often originate in a few small variables on an application program's run-time stack, but if transmission is done using DMA, they must be copied to a special, wired-down metadata area. This proved burdensome; if we were to redesign Hamlyn, we might always send metadata using direct I/O.

6.3 Opening and monitoring connections

There is a "bootstrapping" problem when contacting a long-lived server: a potential client cannot transmit to the server because it has no resources allocated there, and the server cannot send a ticket or seed because it does not know of the client's existence. We considered adding an unreliable FIFO message queue, but we decided that since these operations are not time-critical, they could be done with standard OS services, which might themselves use Hamlyn inside the operating system. (The lowest-level bootstrapping problem here can be solved by allocating well-known slots, one for each remote processor, which

the OS instances can use to establish higher-level communication paths.)

A similar observation applies to detecting peer-process failures. We once thought that an "Are you there?" message should be sent periodically between processes in a highly available system, but if such a polling interval expires without a reply, an application does not know whether the system is overloaded, or the polled host has failed, or a peer process is dead, or the process is stuck in a long computation. On the other hand, the OS has definitive knowledge of process' states and so can prevent much of this confusion. The moral is to let the OS do what it is good at.

6.4 Closing connections

Traditional networks have difficulty providing reliable connection close because of potential message loss. In the absence of such loss, they can do a good job because the OS knows about the connection setup and can tear it down even after the application dies. (This is even true in most application-level protocol suites, which invoke the OS for connection setup/teardown.) In Hamlyn, the OS cannot fulfill this role because it has no knowledge of the connections, so we reverted to a model where our prototype only allows graceful close operations by a sender.

6.5 Stronger security

Hamlyn uses 64-bit protection keys. We estimate that our prototype can detect and discard a packet having an invalid key within 3 μ s. At this rate, a brute-force attack is likely to take about 877,000 years. Keys can be generated using cryptographic-quality pseudo-random number generators, or generators embodying true random processes, so we think that attack by guessing keys is futile. But Hamlyn keys reside in applications' memory address spaces, so our defense against forged messages depends upon memory privacy. For this reason, and because some users find probabilistic protection unsatisfactory, we thought briefly about other techniques:

It would be easy to make protection keys accessible only indirectly, and have applications specify indices into a secure, per-process table of keys, maintained by the OS. A secure OS would then make keys unforgeable. But this scheme would fundamentally alter the Hamlyn paradigm, since almost all communication channel management would then require OS intervention.

In practice, this is largely irrelevant because the main security problem is user passwords, which are much simpler to attack than 64-bit binary keys.

6.6 Interface memory cost

The original Hamlyn design proposed that slots be cached by interface hardware in order to make them abundant and cheap. Our prototype allocates slot data structures in expensive interface card SRAM, and metadata areas in main memory. We never satisfactorily established the right trade-off between function and complexity here.

6.7 Summary

The Hamlyn architecture provides users with a message passing interface having a combined hardware and software latency of just a few microseconds, while providing full protection between mutually suspicious applications. We described the most important techniques underlying our implementation, including design trade-offs that we can make (and have made), and we presented performance measurements.

Our design is optimized for closely-coupled, multicomputer systems. It yields better performance than loosely-coupled clusters of autonomous computers and, due to the inherent isolation of message-passing systems, provides much better fault tolerance than shared-memory systems, as well as inter-application protection at low cost. All of these needs must be addressed if large-scale, parallel machines are to have a significant impact upon general-purpose computing. The Hamlyn architecture is an important step in that direction.

Acknowledgments

Martin Fouts and Bill Worley of HP Laboratories provided the initial encouragement to turn Hamlyn from a random thought into a proper architecture. Monroe Bridges and his colleagues in Hewlett-Packard's Networked Computing Division provided valuable insight from the perspective of product designers, forcing us to justify and simplify our design. Cedric Krumbein and other members of the Network of Workstations project in the Computer Science and Engineering Division of the University of California at Berkeley helped us to design our interface cards, while the staff of Myricom, Inc., helped us to build them and gave us a considerable amount of advice, sample firmware, and early access to their technology, enabling our success.

References

- [Banks93] D. Banks and M. Prudence. A high performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications* 11(2), February 1993.
- [Belady81] L. A. Belady and R. P. Parmelee, and C. A. Scalzi. The IBM history of memory management technology. *IBM Journal of Research and Development*, 25(5):491–503, September 1981.
- [Blumrich94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felton, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, 22(2):142–53. ACM/IEEE, April 1994.
- [Boden95] Nannette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: a Gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.
- [Buzzard95] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. Presented at *HotInterconnects III* (Stanford, CA), August 1995. Available from http://www.hpl.hp.com/personal/John_Wilkes/ftp-index.html#Hamlyn.
- [Clark66] W. A. Clark. The functional structure of OS/360: part III, data management. *IBM Systems Journal*, 5(1):30–51, 1966.
- [Cooper90] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol implementation on the Nectar communication processor. *Proceedings of the ACM SIGCOMM'90 Symposium* (Philadelphia, PA), September 1990.
- [Corbett95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. *3rd Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS'95)* (Santa Barbara, CA), pages 1–15, April 1995.
- [Dalton93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network* 7(4):36–43, July 1993.
- [Davis92] Al Davis. Mayfly: a general-purpose, scalable, parallel processing architecture. *Lisp and Symbolic Computation* 5(1–2):7–48, May 1992.
- [Druschel93] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC), pp 189–202, December 1993.
- [Druschel94] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: a software perspective. *Proceedings of the 1994 ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, pp 2–13, August 1994.
- [IEEE92] Standard for scalable coherent interface (SCI). IEEE Standard 1596-1992.
- [Jacobson95] David M. Jacobson. *Method and apparatus for determining when all packets of a message have*

- arrived. US patent application, filed 24 February 1995, allowed 25 June 1996.
- [Jones96] Rick Jones. *NetPerf*.
<http://www.cup.hp.com/netperf/NetperfPage.html>,
 Hewlett-Packard Company, Cupertino, CA.
- [Karamcheti94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: where does the time go? *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA). ACM, October 1994.
- [Keeton95] Kimberley Keeton, Thomas Anderson, and David Patterson. LogP quantified: the case for low-overhead local area networks. Presented at *HotInterconnects III* (Stanford, CA), August 1995. Available at <http://now.cs.berkeley.edu/Papers/Papers/hotinter95-tcp.ps>.
- [Kranz93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiawicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. *Proceedings of 4th ACM Annual Symposium on Principles and Practice of Parallel Programming*, May 1993.
- [Lumley92] J. Lumley. *A high-throughput network interface for to a RISC workstation*. Hewlett-Packard Laboratories technical report HPL-92-7, January 1992.
- [McKenzie94] Neil R. McKenzie, Kevin Bolding, Carl Ebeling, and Lawrence Snyder. Cranium: an interface for message passing on adaptive routing networks. *Proceedings of Parallel Computer Routing and Communication Workshop* (Seattle, WA), pages 266-80, May 1994.
- [Osborne94] Randy Osborne. A hybrid deposit model for low overhead communication in high speed LANs. *Proc. 4th Intl. IFIP Workshop on Protocols for High-speed Networks*, August 1994. Available as <http://www.merl.com/TR/TR94-02c/Welcome.html>.
- [Reinhardt94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Typhoon and Tempest: user-level shared memory. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, 22(2):325-36. ACM/IEEE, April 1994.
- [Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277-88, November 1984.
- [Stepanov95] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11 (R.1), Hewlett-Packard Laboratories, 1995.
- [Subhlok93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA), May, 1993, pp 13-22.
- [Thekkath94] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. *Proceedings of ASPLOS VI* (4-7 Oct. 1994, San Jose, CA). Published as *Operating Systems Review* 28(5):2-11, December 1994.
- [vonEicken92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schuser. Active Messages: a mechanism for integrated communication and computation. *Proceedings of 19th International Symposium on Computer Architecture* (Gold Coast, Australia), pages 256-66, May 1992.
- [vonEicken94] Thorsten von Eicken, Veena Avula, Anindya Basu and Vineet Buch. Low-latency communication over ATM networks using active messages. Presented at *Hot Interconnects II* (Stanford, CA), August 1994.
- [vonEicken95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net, a user-level interface for parallel and distributed computing. *Proceedings of the ACM Symposium on Operating System Principles* (Copper Mountain Resort, Colorado). Published as *Operating Systems Review* 29(5):40-53, December 1995.
- [Wilkes92] John Wilkes. *Hamlyn—an interface for sender-based communications*. Technical report HPL-OSR-92-13. Operating Systems Research Department, Hewlett-Packard Laboratories, Palo Alto, CA, November 1992. Available from http://www.hpl.hp.com/personal/John_Wilkes/ftp-index.html#Hamlyn.
- [Wilkes95] John Wilkes. *Inter-processor communication system in which messages are stored at locations specified by the sender*. US patent number 5,448,698, granted Sept. 1995.

The authors can be contacted as follows:
 gdb@geoplex.com, {jacobson, marovich, mackey, wilkes}@hpl.hp.com. Please address correspondence to David Jacobson.

Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems

Peter Druschel and Gaurav Banga*

Department of Computer Science
Rice University
Houston, TX 77005

Abstract

The explosive growth of the Internet, the widespread use of WWW-related applications, and the increased reliance on client-server architectures places interesting new demands on network servers. In particular, the operating system running on such systems needs to manage the machine's resources in a manner that maximizes and maintains throughput under conditions of high load. We propose and evaluate a new network subsystem architecture that provides improved fairness, stability, and increased throughput under high network load. The architecture is hardware independent and does not degrade network latency or bandwidth under normal load conditions.

1 Introduction

Most work on operating system support for high-speed networks to date has focused on improving message latency and on delivering the network's full bandwidth to application programs [1, 5, 7, 21]. More recently, researchers have started to look at resource management issues in network servers such as LAN servers, firewall gateways, and WWW servers [16, 17]. This paper proposes a new network subsystem architecture based on *lazy receiver processing (LRP)*, which provides stable overload behavior, fair resource allocation, and increased throughput under heavy load from the network.

State of the art operating systems use sophisticated means of controlling the resources consumed by application processes. Policies for dynamic scheduling, main

memory allocation and swapping are designed to ensure graceful behavior of a timeshared system under various load conditions. Resources consumed during the processing of network traffic, on the other hand, are generally not controlled and accounted for in the same manner. This poses a problem for network servers that face a large volume of network traffic, and potentially spend considerable amounts of resources on processing that traffic.

In particular, UNIX based operating systems and many non-UNIX operating systems use an interrupt-driven network subsystem architecture that gives strictly highest priority to the processing of incoming network packets. This leads to scheduling anomalies, decreased throughput, and potential resource starvation of applications. Furthermore, the system becomes unstable in the face of overload from the network. This problem is serious even with the relatively slow current network technology and will grow worse as networks increase in speed.

We propose a network subsystem architecture that integrates network processing into the system's global resource management. Under this system, resources spent in processing network traffic are associated with and charged to the application process that causes the traffic. Incoming network traffic is scheduled at the priority of the process that receives the traffic, and excess traffic is discarded early. This allows the system to maintain fair allocation of resources while handling high volumes of network traffic, and achieves system stability under overload.

Experiments show that a prototype system based on LRP maintains its throughput and remains responsive even when faced with excessive network traffic on a 155 Mbit/s ATM network. In comparison, a conventional UNIX system collapses under network traffic conditions that can easily arise on a 10 Mbit/s Ethernet. Further results show increased fairness in resource allocation, traffic separation, and increased throughput under high load.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the network subsystem found in

*This work supported in part by National Science Foundation Grant CCR-9503098

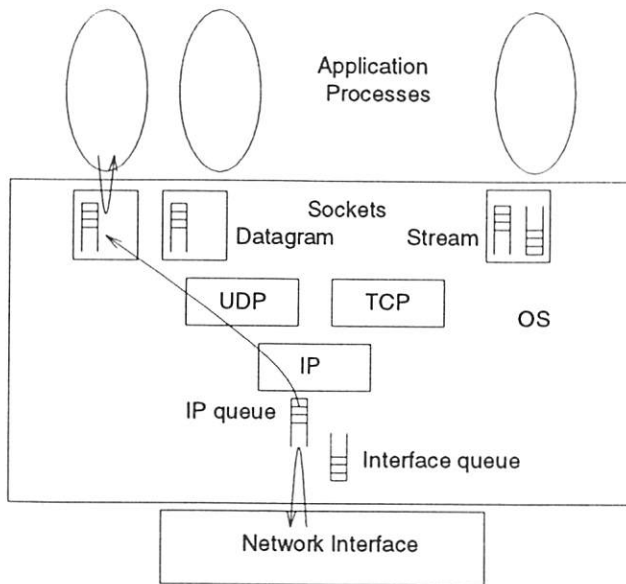


Figure 1: BSD Architecture

BSD UNIX-derived systems [13] and identifies problems that arise when a system of this type is used as a network server. The design of the LRP network architecture is presented in Section 3. Section 4 gives a quantitative performance evaluation of our prototype implementation. Finally, Section 5 covers related work and and Section 6 offers some conclusions.

2 UNIX Network Processing

This section starts with a brief overview of network processing in UNIX operating systems. It then points out problems that arise when a system of this type faces large volumes of network traffic. Finally, we argue that these problems are important by discussing common sources of high network traffic.

To simplify the discussion, we focus on the TCP/UDP/IP protocol suite, and on BSD-derived UNIX systems [13]. Similar problems arise with other protocol suites, in System V-derived UNIX systems, and in many commercial non-UNIX operating systems. Figure 1 illustrates the BSD networking architecture.

2.1 Overview

On the receiving side, the arrival of a network packet is signaled by an interrupt. The interrupt handler, which is part of the network interface device driver, encapsulates the packet in an *mbuf*, queues the packet in the *IP queue*, and posts a software interrupt. In the context of this software interrupt, the packet is processed by IP. After potential reassembly of multiple IP fragments, UDP's or

TCP's input function is called, as appropriate. Finally—still in the context of the software interrupt—the packet is queued on the *socket queue* of the socket that is bound to the packet's destination port. The software interrupt has higher priority than any user process; therefore, whenever a user process is interrupted by a packet arrival, the protocol processing for that packet occurs *before* control returns to the user process. On the other hand, software interrupts have lower priority than hardware interrupts; thus, the reception of subsequent packets can interrupt the protocol processing of earlier packets.

When an application process performs a receive system call¹ on the socket, the packet's data is copied from the *mbufs* into the application's address space. The *mbufs* are then dequeued and deallocated. This final processing step occurs in the context of the user process performing a system call.

On the sending side, data written to a socket by an application is copied into newly allocated *mbufs*. For datagram sockets (UDP), the *mbufs* are then handed to UDP and IP for transmission. After potential fragmentation, the resulting IP packets are then transmitted, or—if the interface is currently busy—placed in the driver's interface queue. All of these actions are executed in the context of the user process that performed the send system call on the socket. Packets queued in the interface queue are removed and transmitted in the context of the network interface's interrupt handler.

For stream sockets (TCP), the *mbufs* are queued in the socket's outgoing socket queue, and TCP's output function is called. Depending on the state of the TCP connection and the arguments to the send call, TCP makes a logical copy of all, some, or none of the queued *mbufs*, processes them for transmission, and calls IP's output function. The resulting IP packets are then transmitted or queued on the interface queue. Again, this processing occurs in the context of the application process performing a system call. As for UDP packets, data is removed from the interface queue and transmitted in the context of the network interface's interrupt handler.

Processing of any remaining data in the socket queue typically occurs in the context of a software interrupt. If TCP receives an acknowledgment, more data from the socket queue may be sent in the context of the software interrupt that was posted to process the incoming acknowledgment. Or, data may be sent in the context of a software interrupt that was scheduled by TCP to indicate a timeout. Data is not removed from the socket queue until its reception was acknowledged by the remote receiver.

CPU time consumed during the processing of network I/O is accounted for as follows. Any processing that

¹We use the term *receive system call* to refer to any of the five system calls available to read data from a socket. The term *send system call* is used analogously to refer to system calls that write data to a socket.

occurs in the context of a user process performing a system call is charged to that process as system time. CPU time spent in software or hardware interrupt handlers is charged to the user process *that was interrupted*. Note that in general, the interrupted process may be unrelated to the network communication that caused the interrupt.

2.2 Problems

We now turn to describe several problems that can arise when a system with conventional network architecture faces high volumes of network traffic. Problems arise because of four aspects of the network subsystem:

Eager receiver processing

Processing of received packets is strictly interrupt-driven, with highest priority given to the capture and storage of packets in main memory; second highest priority is given to the protocol processing of packets; and, lowest priority is given to the applications that consume the messages.

Lack of effective load shedding Packet dropping as a means to resolve receiver overload occurs only after significant host CPU resources have already been invested in the dropped packet.

Lack of traffic separation Incoming traffic destined for one application (socket) can lead to delay and loss of packets destined for another application (socket).

Inappropriate resource accounting CPU time spent in interrupt context during the reception of packets is charged to the application that happens to execute when a packet arrives. Since CPU usage, as maintained by the system, influences a process's future scheduling priority, this is unfair.

Eager receiver processing has significant disadvantages when used in a network server. It gives highest priority to the processing of incoming network packets, regardless of the state or the scheduling priority of the receiving application. A packet arrival will always interrupt a presently executing application, even if any of the following conditions hold true: (1) the currently executing application is not the receiver of the packet; (2) the receiving application is not blocked waiting on the packet; or, (3) the receiving application has lower or equal priority than the currently executing process. As a result, overheads associated with dispatching and handling of interrupts and increased context switching can limit the throughput of a server under load.

Under high load from the network, the system can enter a state known as *receiver livelock* [20]. In this state, the system spends all of its resources processing incoming network packets, only to discard them later because

no CPU time is left to service the receiving application programs. For instance, consider the behavior of the system under increasing load from incoming UDP packets². Since hardware interface interrupt and software interrupts have higher priority than user processes, the socket queues will eventually fill because the receiving application no longer gets enough CPU time to consume the packets. At that point, packets are discarded when they reach the socket queue. As the load increases further, the software interrupts will eventually no longer keep up with the protocol processing, causing the IP queue to fill. The problem is that early stages of receiver processing have strictly higher priority than later stages. Under overload, this causes packets to be dropped only after resources have been invested in them. As a result, the throughput of the system drops as the offered load increases until the system finally spends all its time processing packets only to discard them.

Bursts of packets arriving from the network can cause scheduling anomalies. In particular, the delivery of an incoming message to the receiving application can be delayed by a burst of *subsequently* arriving packets. This is because the network processing of the entire burst of packets must complete before any application process can regain control of the CPU. Also, since all incoming IP traffic is placed in the shared IP queue, aggregate traffic bursts can exceed the IP queue limit and/or exhaust the mbuf pool. Thus, traffic bursts destined for one server process can lead to the delay and/or loss of packets destined for other sockets. This type of traffic interference is generally unfair and undesirable.

2.3 Sources of High Network Load

Network protocols and distributed application programs use flow control mechanisms to prevent a sender process from generating more traffic than the receiver process can handle. Unfortunately, flow control does not necessarily prevent overload of network server machines. Some reasons for this are:

- simultaneous requests from a large number of clients
- misbehaved distributed applications
- incorrect client protocol implementations
- malicious denial-of-service attacks
- broadcast and multicast traffic

TCP connection establishment requests (TCP SYN packets) from a large number of clients can flood a WWW server. This is true despite TCP's flow control mechanism

²Similar problems can arise under load from TCP connection establishment request packets.

(which regulates traffic on established connections) and TCP's exponential backoff strategy for connection establishment requests (which can only limit the rate of retries). The maximal rate of SYN packets is only bounded by the capacity of the network. Similar arguments apply for any server that serves a virtually unlimited client community such as the Internet.

Distributed applications built on top of a simple datagram service such as UDP must implement their own flow and congestion control mechanisms. When these mechanisms are deficient, excessive network traffic can result. Incorrect implementations of flow-controlled protocols such as TCP—not uncommon in the PC market—can have the same effect. The vulnerability of network servers to network traffic overload can be and has been exploited for security attacks³. Thus, current network servers have a protection and security problem, since untrusted application programs running on clients can cause the failure of the shared server.

There are many examples of real-world systems that are prone to the problems discussed above. A packet filtering application-level gateway, such as a firewall, establishes a new TCP connection for every flow that passes through it. An excessive flow establishment rate can overwhelm the gateway. Moreover, a misbehaving flow can get an unfair share of the gateway's resources and interfere with other flows that pass through it. Similar problems can occur in systems that run several server processes, such as Web servers that use a process per connection; or, single process servers that use a kernel thread per connection. Scheduling anomalies, such as those related to bursty data, can be ill-afforded by systems that run multimedia applications. Apart from the above examples, any system that uses eager network processing can be livelocked by an excess of network traffic—this need not always be part of a denial of service attack, and can simply be because of a program error.

These problems make it imperative that a network server be able to control its resources in a manner that ensures efficiency and stability under conditions of high network load. The conventional, interrupt-driven network subsystem architecture does not satisfy this criterion.

3 Design of the LRP Architecture

In this section, we present the design of our network subsystem architecture based on lazy receiver processing (LRP). We start with an overview, and then focus on details of protocol processing for UDP and TCP.

³Often, a denial-of-service attack is used as part of a more elaborate security attack.

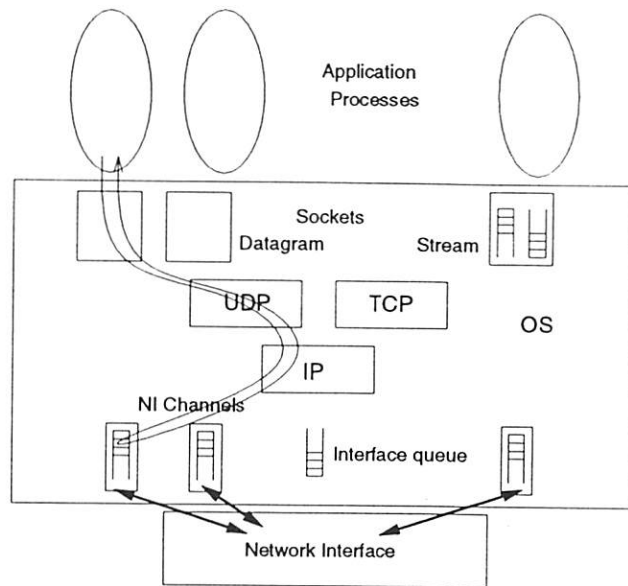


Figure 2: LRP Architecture

The proposed architecture overcomes the problems discussed in the previous section through a combination of techniques: (1) The IP queue is replaced with a per-socket queue that is shared with the network interface (NI). (2) The network interface demultiplexes incoming packets according to their destination socket, and places the packet directly on the appropriate receive queue⁴. Packets destined for a socket with a full receiver queue are silently discarded (early packet discard). (3) Receiver protocol processing is performed at the priority of the receiving process⁵. (4) Whenever the protocol semantics allow it, protocol processing is performed lazily, in the context of the user process performing a receive system call. Figure 2 illustrates the LRP architecture.

There are several things to note about the behavior of this architecture. First, protocol processing for a packet in many cases does not occur until the application requests the packet in a receive system call. Packet processing no longer interrupts the running process at the time of the packet's arrival, unless the receiver has higher scheduling priority than the currently executing process. This avoids inappropriate context switches and can increase performance.

Second, the network interface separates (demultiplexes) incoming traffic by destination socket and places packets directly into per-socket receive queues. Combined with the receiver protocol processing at application

⁴The present discussion assumes that the network interface has an embedded CPU that can be programmed to perform this task. Section 3.2 discusses how LRP can be implemented with an uncooperative NI.

⁵For a shared or multicast socket, this is the highest of the participating processes' priorities.

priority, this provides feedback to the network interface about application processes' ability to keep up with the traffic arriving at a socket. This feedback is used as follows: Once a socket's receive queue fills, the NI discards further packets destined for the socket until applications have consumed some of the queued packets. Thus, the NI can effectively shed load without consuming significant host resources. As a result, the system has stable overload behavior and increased throughput under high load.

Third, the network interface's separation of received traffic, combined with the receiver processing at application priority, eliminates interference among packets destined for separate sockets. Moreover, the delivery latency of a packet cannot be influenced by a subsequently arriving packet of equal or lower priority. And, the elimination of the shared IP queue greatly reduces the likelihood that a packet is delayed or dropped because traffic destined for a different socket has exhausted shared resources.

Finally, CPU time spent in receiver protocol processing is charged to the application process that receives the traffic. This is important since the recent CPU usage of a process influences the priority that the scheduler assigns a process. In particular, it ensures fairness in the case where application processes receive high volumes of network traffic.

Early demultiplexing—a key component of LRP's design—has been used in many systems to support application-specific network protocols [11, 23], to avoid data copying [6, 21], and to preserve network quality-of-service guarantees for real-time communication [10]. Demultiplexing in the network adaptor and multiple NI channels have been used to implement low-latency, high-bandwidth, user-level communication [1, 5]. Protocol processing by user-level threads at application priority has been used in user-level network subsystem implementations [10, 11, 23]. What is new in LRP's design is (1) the lazy, delayed processing of incoming network packets, and (2) the combination and application of the above techniques to provide stability, fairness, and increased throughput under high load. A full discussion of related work is given in Section 5.

It is important to note that the two key techniques used in LRP—lazy protocol processing at the priority of the receiver, and early demultiplexing—are both necessary to achieve stability and fairness under overload. Lazy protocol processing trivially depends on early demultiplexing. To see this, observe that the receiver process of an incoming packet must be known to determine the time and priority at which the packet should be processed.

Conversely, early demultiplexing by itself is not sufficient to provide stability and fairness under overload. Consider a system that combines the traditional eager protocol processing with early demultiplexing. Packets

are dropped immediately in case their destination socket's receive queue is full. One would expect this system to remain stable under overload, since traffic arriving at an overloaded endpoint is discarded early. Unfortunately, the system is still defenseless against overload from incoming packets that do not contain valid user data. For example, a flood of control messages or corrupted data packets can still cause livelock. This is because processing of these packets does not result in the placement of data in the socket queue, thus defeating the only feedback mechanism that can effect early packet discard.

In addition, early demultiplexing by itself lacks LRP's benefits of reduced context switching and fair resource allocation, since it shares BSD's resource accounting and eager processing model. A quantitative comparison of both approaches is given in Section 4. We proceed with a detailed description of LRP's design.

3.1 Sockets and NI Channels

A *network interface (NI) channel* is a data structure that is shared between the network interface and the OS kernel. It contains a receiver queue, a free buffer queue, and associated state variables. The NI determines the destination socket of any received packets and queues them on the receive queue of the channel associated with that socket. Thus, the network interface effectively demultiplexes incoming traffic to their destination sockets.

When a socket is bound to a local port (either implicitly or explicitly by means of a `bind()` system call), an NI channel is created. Also, when a connected stream socket is created, it is allocated its own NI channel. Multiple sockets bound to the same UDP multicast group share a single NI channel. All traffic destined for or originating from a socket passes through that socket's NI channel.

3.2 Packet Demultiplexing

LRP requires that the network interface be able to identify the destination socket of an incoming network packet, so that the packet can be placed on the correct NI channel. Ideally, this function should be performed by the NI itself. Incidentally, many commercial high-speed network adaptors contain an embedded CPU, and the necessary demultiplexing function can be performed by this CPU. We call this approach LRP with *NI demux*. In the case of network adaptors that lack the necessary support (e.g., inexpensive Fast Ethernet adaptors), the demultiplexing function can be performed in the network driver's interrupt handler. We call this approach *soft demux*. Here, some amount of host interrupt processing is necessary to demultiplex incoming packets. Fortunately, with current technology, this overhead appears to be small enough to still maintain good stability under overload. The advan-

tage of this approach is that it will work with any network adaptor, i.e., it is hardware independent. We will quantitatively evaluate both demultiplexing approaches in Section 4.

Our demultiplexing function is self-contained, and has minimal requirements on its execution environment (non-blocking, no dynamic memory allocation, no timers). As such, it can be readily integrated in a network interface's firmware, or the device's host interrupt handler. The function can efficiently demultiplex all packets in the TCP/IP protocol family, including IP fragments. In rare cases, an IP fragment does not contain enough information to allow demultiplexing to the correct endpoint. This happens when the fragment containing the transport header of a fragmented IP packet does not arrive first. In this case, the offending packet is placed on a special NI channel reserved for this purpose. The IP reassembly function checks this channel queue when it misses fragments during reassembly.

Throughout this paper, whenever reference is made to actions performed by the network interface, we mean that the action is performed either by the NI processor (in the case of NI demux), or the host interrupt handler (in the case of soft demux).

3.3 UDP protocol processing

For unreliable, datagram-oriented protocols like UDP, network processing proceeds as follows: The transmit side processing remains largely unchanged. Packets are processed by UDP and IP code in the context of the user process performing the send system call. Then, the resulting IP packet(s) are placed on the interface queue.

On the receiving side, the network interface determines the destination socket of incoming packets and places them on the corresponding channel queue. If that queue is full, the packet is discarded. If the queue was previously empty, and a state flag indicates that interrupts are requested for this socket, the NI generates a host interrupt⁶. When a user process calls a receive system call on a UDP socket, the system checks the associated channel's receive queue. If the queue is non-empty, the first packet is removed; else, the process is blocked waiting for an interrupt from the NI. After removing a packet from the receive queue, IP's input function is called, which will in turn call UDP's input function. Eventually the processed packet is copied into the application's buffer. All these steps are performed in the context of the user process performing the system call.

There are several things to note about the receiver processing. First, protocol processing for a packet does not occur until the application is waiting for the packet, the packet has arrived, and the application is scheduled to run.

⁶With soft demux, a host interrupt always occurs upon packet arrival.

As a result, one might expect reduced context switching and increased memory access locality. Second, when the rate of incoming packets exceeds the rate at which the receiving application can consume the packets, the channel receive queue fills, causing the network interface to drop packets. This dropping occurs before significant host resources have been invested in the packet. As a result, the system has good overload behavior: As the offered rate of incoming traffic approaches the capacity of the server, the throughput reaches its maximum and stays at its maximum even if the offered rate increases further⁷.

It is important to realize that LRP does *not* increase the latency of UDP packets. The only condition under which the delivery delay of a UDP packet could increase under LRP is when a host CPU is idle between the time of arrival of the packet and the invocation of the receive system call that will deliver the packet to the application. This case can occur on multiprocessor machines, and on a uniprocessor when the only runnable application blocks on an I/O operation (e.g., disk) before invoking the receive system call. To eliminate this possibility, an otherwise idle CPU should always perform protocol processing for any received packets. This is easily accomplished by means of a kernel thread with minimal priority that checks NI channels and performs protocol processing for any queued UDP packets.

3.4 TCP protocol processing

Protocol processing is slightly more complex for a reliable, flow-controlled protocol such as TCP. As in the original architecture, data written by an application is queued in the socket queue. Some data may be transmitted immediately in the context of the user process performing the send system call. The remaining data is transmitted in response to arriving acknowledgments, and possibly in response to timeouts.

The main difference between UDP and TCP processing in the LRP architecture is that receiver processing cannot be performed only in the context of a receive system call, due to the semantics of TCP. Because TCP is flow controlled, transmission of data is paced by the receiver via acknowledgments. Achieving high network utilization and throughput requires timely processing of incoming acknowledgments. If receiver processing were performed only in the context of receive system calls, then at most one TCP congestion window of data could be transmitted between successive receive system calls, resulting in poor performance for many applications.

The solution is to perform receiver processing for TCP sockets asynchronously when required. Packets arriving on TCP connections can thus be processed even when

⁷With soft demux, the throughput diminishes slightly as the offered load increases, due to the demultiplexing overhead.

the application process is not blocked on a receive system call. Unlike in conventional architectures, this asynchronous protocol processing does not take strict priority over application processing. Instead, the processing is scheduled at the priority of the application process that uses the associated socket, and CPU usage is charged back to that application⁸. Under normal conditions, the application has a sufficiently high priority to ensure timely processing of TCP traffic. If an excessive amount of traffic arrives at the socket, the application's priority will decay as a result of the high CPU usage. Eventually, the protocol processing can no longer keep up with the offered load, causing the channel receiver queue to fill and packets to be dropped by the NI. In addition, protocol processing is disabled for listening sockets that have exceeded their listen backlog limit, thus causing the discard of further SYN packets at the NI channel queue. As shown in Section 4, TCP sockets enjoy similar overload behavior and traffic separation as UDP sockets under LRP.

There are several ways of implementing asynchronous protocol processing (APP). In systems that support (kernel) threads (i.e., virtually all modern operating systems), an extra thread can be associated with application processes that use stream (TCP) sockets. This thread is scheduled at its process's priority and its CPU usage is charged to its process. Since protocol processing always runs to completion, no state needs to be retained between activations. Therefore, it is not necessary to assign a private runtime stack to the APP thread; a single per CPU stack can be used instead. The resulting per-process space overhead of APP is one thread control block. This overhead can be further reduced through the use of continuations [3]. The exact choice of a mechanism for APP greatly depends on the facilities available in a particular UNIX kernel. In our current prototype implementation, a kernel process is dedicated to TCP processing.

3.5 Other protocol processing

Processing for certain network packets cannot be directly attributed to any application process. In the TCP/IP suite, this includes processing of some ARP, RARP, ICMP packets, and IP packet forwarding. In LRP, this processing is charged to daemon processes that act as proxies for a particular protocol. These daemons have an associated NI channel, and packets for such protocols are demultiplexed directly onto the corresponding channel. For example, an IP forwarding daemon is charged for CPU time spent on forwarding IP packets, and its priority con-

trols resources spent on IP forwarding⁹. The IP daemon competes with other processes for CPU time.

4 Performance

In this section, we present experiments designed to evaluate the effectiveness of the LRP network subsystem architecture. We start with a description of the experimental setup and the prototype implementation, and proceed to present the results of various experiments.

4.1 Experimental Setup

All experiments were performed on Sun Microsystems SPARCstation 20 model 61 workstations (60MHz SuperSPARC+, 36KB L1, 1MB L2, SPECint92 98.2). The workstations are equipped with 32MB of memory and run SunOS 4.1.3.U1. A 155 Mbit/s ATM local area network connects the workstations, using FORE Systems SBA-200 network adaptors. These network adaptors include an Intel i960 processor that performs cell fragmentation and reassembly of protocol data units (PDUs). Note that LRP does not depend on a specific network adaptor or ATM networks. SOFT-LRP can be used with any network and NI.

The LRP architecture was implemented as follows. We modified the TCP/UDP/IP network subsystem that comes with the 4.4 BSD-Lite distribution [24] to optionally implement LRP. The resulting code was then downloaded into the SunOS kernel as a loadable kernel module and attached to the socket layer as a new protocol family (PF.LRP). A custom device driver was developed for the FORE network adaptor. The 4.4 BSD-Lite networking subsystem was used because of its performance and availability in source form. (We did not have access to SunOS source code.) The 4.4 BSD networking code was slightly modified to work with SunOS mbufs. At the time of this writing, the prototype implementation uses a kernel process to perform asynchronous protocol processing for TCP.

Since we were unable to obtain source code for the SBA-200 firmware, we could not integrate our own demultiplexing function in this network adaptor. However, we know enough about the interface's architecture to be confident that the function could be easily integrated, given the source code. To evaluate packet demultiplexing in the network adaptor (NI demux), we used instead the SBA-200 firmware developed by Cornell University's U-Net project [1]. This firmware performs demultiplexing based on the ATM virtual circuit identifier (VCI). A signaling scheme was used that ensures that a separate

⁸In UNIX, more than one process can wait to read from a socket. In this case, the process with the highest priority performs the protocol processing.

⁹QoS attributes or IPv6 flows could be used in an LRP based IP gateway to provide more fine-grained resource control.

ATM VCI is assigned for traffic terminating or originating at each socket. The resulting implementation of NI-LRP is fully functional.

4.2 Experimental Results

All experiments were performed on a private ATM network between the SPARCstations. The machines were running in multiuser mode, but were not shared by other users.

The first experiment is a simple test to measure UDP latency and throughput, and TCP throughput. Its purpose is to demonstrate that the LRP architecture is competitive with traditional network subsystem implementations in terms of these basic performance criteria. Moreover, we include the results for an unmodified SunOS kernel with the Fore ATM device driver for comparison. Latency was measured by ping-ponging a 1-byte message between two workstations 10,000 times, measuring the elapsed time and dividing to obtain round-trip latency. UDP throughput was measured using a simple sliding-window protocol (UDP checksumming was disabled.) TCP throughput was measured by transferring 24 Mbytes of data, with the socket send and receive buffers set to 32 KByte. Table 1 shows the results.

The numbers clearly demonstrate that LRP's basic performance is comparable with the unmodified BSD system from which it was derived. That is, LRP's improved overload behavior does not come at the cost of low-load performance. Furthermore, both BSD and LRP with our device driver perform significantly better than SunOS with the Fore ATM driver in terms of latency and UDP bandwidth. This is due to performance problems with the Fore driver, as discussed in detail in [1].

SunOS exhibits a performance anomaly that causes its base round-trip latency—as measured on otherwise idle machines—to drop by almost 300 μ secs, when a compute-bound background process is running on both the client and the server machine. We have observed this effect in many different tests with SunOS 4.1.3_U1 on the SPARCstation 20. The results appear to be consistent with our theory that the cost of dispatching a hardware/software interrupt and/or the receiver process in SunOS depends on whether the machine is executing the idle loop or a user process at the time a message arrives from the network. Without access to source code, we were unable to pinpoint the source of this anomaly.

Since our modified systems (4.BSD, NI-LRP, SOFT-LRP) are all based on SunOS, they were equally affected by this anomaly. Apart from affecting the base round-trip latency, the anomaly can perturb the results of tests with varying rates and concurrency of network traffic, since these factors influence the likelihood that an incoming packet interrupts a user process. To eliminate this vari-

able, some of the experiments described below were run with low-priority, compute-bound processes running in the background, to ensure that incoming packets never interrupt the idle loop.

The next experiment was designed to test the behavior of the LRP architecture under overload. In this test, a client process sends short (14 byte) UDP packets to a server process on another machine at a fixed rate. The server process receives the packets and discards them immediately. Figure 3 plots the rate at which packets are received and consumed by the server process as a function of the rate at which the client transmits packets.

With the conventional 4.4 BSD network subsystem, the throughput increases with the offered load up to a maximum of 7400 pkts/sec. As the offered load increases further, the throughput of the system decreases, until the system approaches livelock at approximately 20,000 pkts/sec. With NI-LRP, on the other hand, throughput increases up to the maximum of 11,000 pkts/sec and remains at that rate as the offered load increases further. This confirms the effectiveness of NI-LRP's load shedding in the network interface, before any host resources have been invested in handling excess traffic. Instrumentation shows that the slight drop in NI-LRP's delivery rate beyond 19,000 pkts/sec is actually due to a reduction in the delivery rate of our ATM network, most likely caused by congestion-related phenomena in either the switch or the network interfaces.

SOFT-LRP refers to the case where demultiplexing is performed in the host's interrupt handler (soft demux). The throughput peaks at 9760 pkts/sec, but diminishes slightly with increasing rate due to the overhead of demultiplexing packets in the host's interrupt handler. This confirms that, while NI-LRP eliminates the possibility of livelock, SOFT-LRP merely postpones its arrival. However, on our experimental ATM network hardware/software platform, we have been unable to generate high enough packet rates to cause livelock in the SOFT-LRP kernel, even when using an in-kernel packet source on the sender.

For comparison, we have also measured the overload behavior of a kernel with early demultiplexing only (Early-Demux). The system performs demultiplexing in the interrupt handler (as in SOFT-LRP), drops packets whose destination socket's receiver queue is full, and otherwise schedules a software interrupt to process the packet. Due to the early demultiplexing, UDP's PCB lookup was bypassed, as in the LRP kernels. The system displays improved stability under overload compared with BSD, a result of early packet discard. The rate of decline under overload is comparable to that of SOFT-LRP, which is consistent with their use of the same demultiplexing mechanism. However, the throughput of the Early-Demux kernel is only between 40–65% of SOFT-

System	round-trip latency (μ secs)	UDP throughput (Mbps)	TCP throughput (Mbps)
SunOS, Fore driver	1006	64	63
4.4 BSD	855	82	69
LRP (NI Demux)	840	92	67
LRP (Soft Demux)	864	86	66

Table 1: Throughput and Latency

Rate Delivered to Application (pkts/sec)

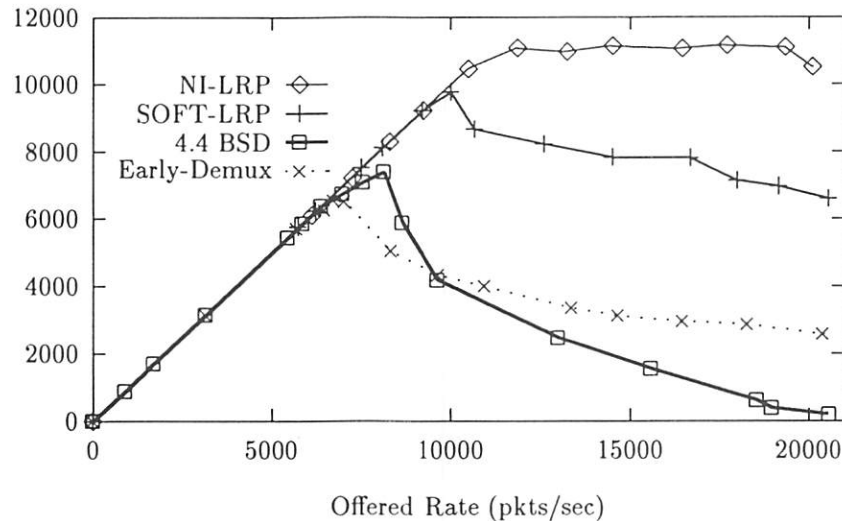


Figure 3: Throughput versus offered load

LRP's throughput across the overload region.

Both variants of LRP display significantly better throughput than both the conventional 4.4 BSD system, and the Early-Demux kernel. The maximal delivered rate of NI-LRP is 51% and that of SOFT-LRP is 32% higher than BSD's maximal rate (11163 vs. 9760 vs. 7380 pkts/sec). Note that the throughput with SOFT-LRP at the maximal offered rate is within 12% of BSD's maximal throughput.

In order to understand the reasons for LRP's throughput gains, we instrumented the kernels to capture additional information. It was determined that the Maximum Loss Free Receive Rate (MLFRR) of SOFT-LRP exceeded that of 4.4BSD by 44% (9210 vs. 6380 pkts/sec). 4.4BSD and LRP drop packets at the socket queue or NI channel queue, respectively, at offered rates beyond their MLFRR. 4.4BSD additionally starts to drop packets at the IP queue at offered rates in excess of 15,000 pkts/sec. No packets were dropped due to lack of mbufs.

Obviously, early packet discard does not play a role in any performance differences at the MLFRR. With the exception of demultiplexing code (early demux in LRP versus PCB lookup in BSD) and differences in the device

driver code, all four kernels execute the same 4.4BSD networking code. Moreover, the device driver and demultiplexing code used in Early-Demux and SOFT-LRP are identical, eliminating these factors as potential contributors to LRP's throughput gains. This suggests that the performance gains in LRP must be due in large part to factors such as reduced context switching, software interrupt dispatch, and improved memory access locality.

Our next experiment measures the latency that a client experiences when contacting a server process on a machine with high network load. The client, running on machine A, ping-pongs a short UDP message with a server process (ping-pong server) running on machine B. At the same time, machine C transmits UDP packets at a fixed rate to a separate server process (blast server) on machine B, which discards the packets upon arrival. Figure 4 plots the round-trip latency experienced by the client as a function of the rate at which packets are transmitted from machine C to the blast server (background load). To avoid the abovementioned performance anomaly in SunOS, the machines involved in the ping-pong exchange were each running a low-priority (nice +20) background process executing an infinite loop.

Round-trip Latency (microseconds)

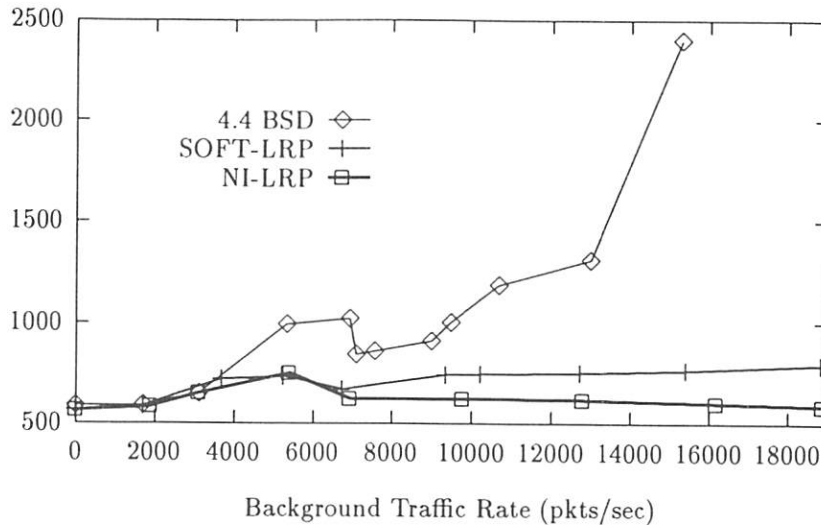


Figure 4: Latency with concurrent load

In all three systems, the measured latency varies with the background traffic rate. This variation is caused by the arrival of background traffic packets during the software processing of a ping-pong packet on the receiver. Arrivals of background traffic delay the processing of the request and/or the transmission of the response message, thus causing an increase in the round-trip delay. The magnitude of this delay is determined by two factors: The rate of arrivals, and the length of the interruptions caused by each arrival. This length of interruptions consists of the fixed interrupt processing time (hardware interrupt in LRP, hardware plus software interrupt in BSD), plus the optional time for scheduling of the blast server, and delivery of the message. This last component only occurs when the blast server's priority exceeds that of the ping-pong server, i.e., it is a function of SunOS's scheduling policy.

Instrumentation and modeling confirmed that the two main factors shaping the graphs are (1) the length of the fixed interrupt processing and (2) the scheduling-dependent overhead of delivering messages to the blast receiver. The fixed interrupt overhead causes a non-linear increase in the latency as the background traffic rises. Due to the large overhead (hardware plus software interrupt, including protocol processing, approximately $60\mu\text{secs}$), the effect is most pronounced in 4.4BSD. SOFT-LRP's reduced interrupt overhead (hardware interrupt, including demux, approx. $25\mu\text{secs}$), results in only a gradual increase. With NI-LRP (hardware interrupt with minimal processing), this effect is barely noticeable.

The second factor leads to an additional increase in latency at background traffic rates up to 7000 pkts/sec.

The UNIX scheduler assigns priorities based on a process's recent CPU usage. As a result, it tends to favor a process that had been waiting for the arrival of a network packet, over the process that was interrupted by the packet's arrival. At low rates, the blast receiver is always blocked when a blast packet arrives. If the arrival interrupts the ping-pong server, the scheduler will almost always give the CPU to the blast receiver, causing a substantial delay of the ping-pong message. At rates around 6000 pkts/sec, the blast receiver is nearing saturation, thus turning compute-bound. As a result, its priority decreases, and the scheduler now preferentially returns the CPU to the interrupted ping-pong server immediately, eliminating this effect at high rates.

The additional delay caused by context switches to the blast server is much stronger in BSD as in the two LRP systems (1020 vs. $750\mu\text{secs}$ peak). This is caused by the mis-accounting of network processing in BSD. In that system, protocol processing of blast messages that arrive during the processing of a ping-pong message is charged to the ping-pong server process. This depletes the priority of the ping-pong server, and increases the likelihood that the scheduler decides to assign the CPU to the blast server upon arrival of a message. Note that in a system that supports fixed-priority scheduling (e.g., Solaris), the influence of scheduling could be eliminated by assigning the ping-pong server statically highest priority. The result is nevertheless interesting in that it displays the effect of CPU mis-accounting on latency in a system with a dynamic scheduling policy.

With BSD, packet dropping at the IP queue makes latency measurements impossible at rates beyond 15,000

RPC	System	Worker elapsed time (secs)	Server (RPCs/sec)
Fast	4.4BSD	49.7	3120
	SO-LRP	38.7	3133
	NI-LRP	34.6	3410
Medium	4.4BSD	47.1	2712
	SO-LRP	37.9	2759
	NI-LRP	34.1	2783
Slow	4.4BSD	43.9	2045
	SO-LRP	38.5	2134
	NI-LRP	35.7	2208

Table 2: Synthetic RPC Server Workload

pkts/sec. In the LRP systems, no dropped latency packets were observed, which is due to LRP's traffic separation.

Our next experiment attempts to more closely model a mix of workloads typical for network servers. Three processes run on a server machine. The first server process, called the worker, performs a memory-bound computation in response to an RPC call from a client. This computation requires approximately 11.5 seconds of CPU time and has a memory working set that covers a significant fraction (35%) of the second level cache. The remaining two server processes perform short computations in response to RPC requests. A client on the other machine sends an RPC request to the worker process. While the worker RPC is outstanding, the client sends RPC requests to the remaining server processes in such a way that (1) each server has a number of outstanding RPC requests at all times, and (2) the requests are distributed near uniformly in time. (1) ensures that the RPC server processes never block on receiving from the network¹⁰. The purpose of (2) is to make sure there is no correlation between the scheduling of the server processes, and the times at which requests are issued by the client. Note that in this test, the clients generate requests at the maximal throughput rate of the server. That is, the server is not operating under conditions of overload. The RPC facility we used is based on UDP datagrams.

Table 2 shows the results of this test. The total elapsed time for completion of the RPC to the worker process is shown in the third column. The rightmost column shows the rate at which the servers process RPCs, concurrently with each other and the worker process. "Fast", "Medium" and "Slow" correspond to tests with different amounts of per-request computations performed in the two RPC server processes. In each of the tests, the server's throughput (considering rate of RPCs completed and worker completion time) is lowest with BSD, higher

¹⁰This is to ensure that the UNIX scheduler does not consider these server processes I/O-bound, which would tend to give them higher scheduling priority.

with SOFT-LRP (SO-LRP), and highest with NI-LRP. In the "Medium" case, where the RPC rates are within 3% for each of the systems, the worker completion time with SOFT-LRP is 20% lower, and with NI-LRP 28% lower than with BSD. In the "Fast" case, NI-LRP achieves an almost 10% higher RPC rate and a 30% lower worker completion time than BSD. This confirms that LRP-based servers have increased throughput under high load. Note that packet discard is not a factor in this test, since the system is not operating under overload. Therefore, reduced context switching and improved locality must be responsible for the higher throughput with LRP.

Furthermore, the LRP systems maintain a fair allocation of CPU resources under high load. With SOFT-LRP and NI-LRP, the worker process's CPU share (CPU time / elapsed completion time) ranges from 29% to 33%, which is very close to the ideal 1/3 of the available CPU, compared to 23%–26% with BSD. This demonstrates the effect of mis-accounting in BSD, which tends to favor processes that perform intensive network communication over those that do not. Observe that this effect is distinct from, and independent of, the UNIX scheduler's tendency to favor I/O-bound processes.

Finally, we conducted a set of experiments with a real server application. We configured a machine running a SOFT-LRP kernel as a WWW server, using the NCSA httpd server software, revision 1.5.1. A set of informal experiments show that the server is dramatically more stable than a BSD based server under overload. To test this, Mosaic clients were contacting the server, while a test program running on a third machine was sending packets to a separate port on the server machine at a high rate (10,000 packets/sec). An HTTP server based on 4.4 BSD freezes completely under these conditions, i.e., it no longer responds to any HTTP requests, and the server console appears dead. With LRP and soft demux, the server responds to HTTP requests and the console is responsive, although some increase in response time is noticeable.

The results of a quantitative experiment are shown in Figure 5. In this test, eight HTTP clients on a single machine continually request HTTP transfers from the server. The requested document is approximately 1300 bytes long. The eight clients saturate the HTTP server. A second client machine sends fake TCP connection establishment requests (SYN packets) to a dummy server running on the server machine that also runs the HTTP server. No connections are ever established as a result of these requests; TCP on the server side discards most of them once the dummy server's listen backlog is exceeded. To avoid known performance problems with BSD's PCB lookup function in HTTP servers [16], the TCP TIME_WAIT period was set to 500ms, instead of the default 30 seconds. The test were run for long periods

HTTP transfers per second

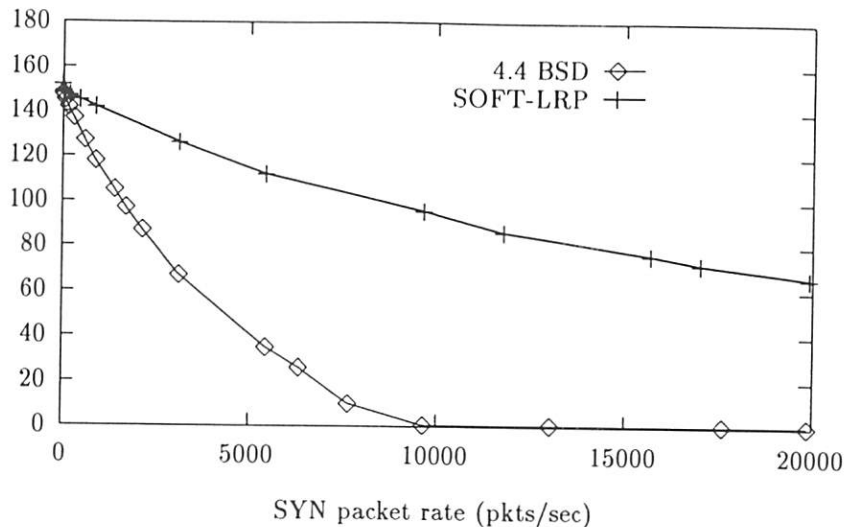


Figure 5: HTTP Server Throughput

of time to ensure steady-state behavior. Furthermore, the LRP system performed a redundant PCB lookup to eliminate any bias due to the greater efficiency of the early demultiplexing in LRP. Note that the results of this test were not affected by the TCP bug described in RFC 1948.

The graphs show the number of HTTP transfers completed by all clients, as a function of the rate of SYN packets to the dummy server, for 4.4 BSD and SOFT-LRP. The throughput of the 4.4 BSD-based HTTP server sharply drops as the rate of background requests increases, entering livelock at close to 10,000 SYN pkts/sec. The reason is that BSD's processing of SYN packets in software interrupt context starves the httpd server processes for CPU resources. Additionally, at rates above 6400 SYN pkts/sec, packets are dropped at BSD's shared IP queue. This leads to the loss of both TCP connection requests from real HTTP client and traffic on established TCP connections. Lost TCP connection requests cause TCP on the client side to back off exponentially. Lost traffic on established connections cause TCP to close its congestion window. However, the dominant factor in BSD's throughput decline appears to be the starvation of server processes.

With LRP, the throughput decreases relatively slowly. At a rate of 20,000 background requests per second, the LRP server still operates at almost 50% of its maximal throughput¹¹. With LRP, traffic on each established TCP connection, HTTP connection requests, and dummy re-

quests are all demultiplexed onto separate NI channels and do not interfere. As a result, traffic to the dummy server does not cause the loss of HTTP traffic at all. Furthermore, most dummy SYN packets are discarded early at the NI channel queue. The predominant cause of the decline in the SOFT-LRP based server's throughput is the overhead of software demultiplexing.

It should be noted that, independent of the use of LRP, an Internet server must limit the number of active connections to maintain stability. A related issue is how well LRP works with a large number of established connections, as has been observed on busy Internet servers [15]. SOFT-LRP uses one extra mbuf compared to 4.4BSD for each established TCP connection, so SOFT-LRP should scale well to large numbers of active connections. NI-LRP, on the other hand, dedicates resources on the network interface for each endpoint and is not likely to scale to thousands of allocated NI channels. However, most of the established connections on a busy web server are in the TIME_WAIT state. This can be exploited by deallocating an NI channel as soon as the associated TCP connection enters the TIME_WAIT state. Any subsequently arriving packets on this connection are queued at a special NI channel which is checked by TCP's *slow-timo* code. Since such traffic is rare, this does not affect NI-LRP's behavior in the normal case.

5 Related Work

Experiences with DEC's 1994 California Election HTTP server reveal many of the problems of a conventional network subsystem architecture when used as a busy HTTP

¹¹Note that a (slow) T1 link is capable of carrying almost 5000 SYN packets per second. With the emerging faster network links, routers, and a sufficiently large user community, a server could easily be subjected to such rates.

server [15]. Mogul [16] suggests that novel OS support may be required to satisfy the needs of busy servers.

Mogul and Ramakrishnan [17] devise and evaluate a set of techniques for improving the overload behavior of an interrupt-driven network architecture. These techniques avoid receiver livelock by temporarily disabling hardware interrupts and using polling under conditions of overload. Disabling interrupts limits the interrupt rate and causes early packet discard by the network interface. Polling is used to ensure progress by fairly allocating resources among receive and transmit processing, and multiple interfaces.

The overload stability of their system appears to be comparable to that of NI-LRP, and it has an advantage over SOFT-LRP in that it eliminates—rather than postpones—livelock. On the other hand, their system does not achieve traffic separation, and therefore drops packets irrespective of their destination during periods of overload. Their system does not attempt to charge resources spent in network processing to the receiving application, and it does not attempt to reduce context switching by processing packets lazily. A direct quantitative comparison between LRP and their system is difficult, because of differing hardware/software environments and benchmarks.

Many researchers have noted the importance of *early demultiplexing* to high-performance networking. Demultiplexing immediately at the network interface point is necessary for maintaining network quality of service (QoS) [22], it enables user-level implementations of network subsystems [2, 7, 11, 21, 23], it facilitates copy-avoidance by allowing smart placement of data in main memory [1, 2, 5, 6], and it allows proper resource accounting in the network subsystem [14, 19]. This paper argues that early demultiplexing also facilitates fairness and stability of network subsystems under conditions of overload. LRP uses early demultiplexing as a key component of its architecture.

Packet filters [12, 18, 25] are mechanisms that implement early demultiplexing without sacrificing layering and modularity in the network subsystem. In the most recent incarnations of packet filters, dynamic code generation is used to eliminate the overhead of the earlier interpreted versions [8].

Architecturally, the design of LRP is related to *user-level network subsystems*. Unlike LRP, the main goal of these prior works is to achieve low communication latency and high bandwidth by removing protection boundaries from the critical send/receive path, and/or by enabling application-specific customization of protocol services. To the best of our knowledge, the behavior of user-level network subsystems under overload has not been studied.

U-Net [1] and Application Device Channels (ADC)

[4, 5] share with NI-LRP the approach of using the network interface to demultiplex incoming packets and placing them on queues associated with communication endpoints. With U-Net and ADCs, the endpoint queues are mapped into the address space of application processes. More conventional user-level networking subsystems [7, 11, 23] share with SOFT-LRP the early demultiplexing of incoming packets by the OS kernel (software). Demultiplexed packets are then handed to the appropriate application process using an upcall. In all user-level network subsystems, protocol processing is performed by user-level threads. Therefore, network processing resources are charged to the application process and scheduled at application priority.

Based on the combination of early demultiplexing and protocol processing by user-level threads, user-level network subsystems can be in principle expected to display improved overload stability. Since user-level threads are normally prioritized to compete with other user and kernel threads, protocol processing cannot starve other applications as in BSD. A user-level network subsystem's resilience to livelock depends then on the overhead of packet demultiplexing on the host. When demultiplexing and packet discard are performed by the NI as in [1, 5], the system should be free of livelock. When these tasks are performed by the OS kernel as in [7, 11, 23], the rate at which the system experiences livelock depends on the overhead of packet demultiplexing (as in SOFT-LRP). Since the systems described in the literature use interpreted packet filters for demultiplexing, the overhead is likely to be high, and livelock protection poor. User-level network subsystems share with LRP the improved fairness in allocating CPU resources, because protocol processing occurs in the context of the receiver process.

User-level network subsystems allow applications to use application-specific protocols on top of the raw network interface. The performance (i.e., latency, throughput) of such protocols under overload depends strongly on their implementation's processing model. LRP's technique of delaying packet processing until the application requests the associated data can be applied to such protocols. The following discussion is restricted to user-level implementations of TCP/IP.

The user-level implementations of TCP/IP described in the literature share with the original BSD architecture the eager processing model. That is, a dedicated user thread (which plays the role of the BSD software interrupt) is scheduled as soon as a packet arrives, regardless of whether or not the application is waiting for the packet. As in BSD, this eager processing can lead to additional context switching, when compared to LRP.

The single shared IP queue in BSD is replaced with a per-application IP queue that is shared only among multiple sockets in a single application. As a result, the

system ensures traffic separation among traffic destined for different applications, but not necessarily among traffic destined for different sockets within a single application. Depending on the thread scheduling policy and the relative priority of the dedicated protocol processing thread(s) and application thread(s), it is possible that incoming traffic can cause an application process to enter a livelock state, where the network library thread consumes all CPU resources allocated to the application, with no CPU time left for the application threads. Traffic separation and livelock protection within an application process are important, for instance, in single-process HTTP servers.

Finally, UNIX based user-level TCP/IP implementations revert to conventional network processing under certain conditions (e.g., whenever a socket is shared among multiple processes.) In this case, the system's overload behavior is similar to that of a standard BSD system.

In summary, we expect that user-level network implementations—while designed with different goals in mind—share some but not all of LRP's benefits with respect to overload. This paper identifies and evaluates techniques for stability, fairness, and performance under overload, independent of the placement of the network subsystem (application process, network server, or kernel). We fully expect that LRP's design principles can be applied to improve the overload behavior of kernelized, server-based, and user-level implementations of network subsystems.

Livelock and other negative effects of BSD's interrupt-driven network processing model can be viewed as an instance of a priority inversion problem. The real-time OS community has developed techniques for avoiding priority inversion in communication systems in order to provide quality of service guarantees for real-time data streams [9, 10]. RT-Mach's network subsystem [10], which is based on the Mach user-level network implementation [11], performs early demultiplexing, and then hands incoming packets for processing to a real-time thread with a priority and resource reservation appropriate for the packet's stream. Like LRP, the system employs early demultiplexing, schedules protocol processing at a priority appropriate to the data's receiver, and charges resources to the receiver. Unlike LRP, it does not attempt to delay protocol processing until the data is requested by the application. Moreover, the overhead of the Mach packet filter is likely to make RT-Mach vulnerable to overload. We fully expect that LRP, when combined with real-time thread scheduling, is applicable to real-time networking, without requiring user-level protocols.

6 Conclusion

This paper introduces a novel network subsystem architecture suitable for network server systems. Performance evaluations indicate that under conditions of high load, the architecture offers increased throughput, stable overload behavior, and reduced interference among traffic destined for separate communication endpoints.

More specifically, LRP's lazy, delayed processing of received network packets reduces context switching and can result in increased server throughput under high load. LRP's combination of early packet demultiplexing, early packet discard, and the processing of incoming network packets at the receiver's priority provide improved traffic separation and stability under overload.

A public release of our SunOS-based prototype is planned for the Fall of 1996. The source code, along with additional technical information can be found at "<http://www.cs.rice.edu/CS/Systems/LRP/>".

Acknowledgments

We are indebted to our OSDI shepherd Jeff Mogul and the anonymous reviewers, whose comments have helped to improve this paper. Also, thanks to Thorsten von Eicken and the U-Net group at Cornell for making the U-Net NI firmware available to us.

References

- [1] A. Bas, V. Buch, W. Vogels, and T. von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 40–53, 1995.
- [2] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium*, Palo Alto, CA, Aug. 1995.
- [3] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.
- [4] P. Druschel. Operating systems support for high-speed networking. Technical Report TR 94-24, Department of Computer Science, University of Arizona, Oct. 1994.

- [5] P. Druschel, B. S. Davie, and L. L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Conference*, pages 2–13, London, UK, Aug. 1994.
- [6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Dec. 1993.
- [7] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s LAN. In *Proceedings of the SIGCOMM '94 Conference*, pages 14–23, London, UK, Aug. 1994.
- [8] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the SIGCOMM '96 Conference*, pages 53–59, Palo Alto, CA, Aug. 1996.
- [9] K. Jeffay. On Latency Management in Time-Shared Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 86–90, Seattle, WA, May 1994.
- [10] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. In *the proceedings of IEEE Real-time Technology and Applications Symposium*, June 1996.
- [11] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.
- [12] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX '93 Winter Conference*, pages 259–269, Jan. 1993.
- [13] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.
- [14] J. C. Mogul. Personal communication, Nov. 1992.
- [15] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.
- [16] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.
- [17] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.
- [18] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Nov. 1987.
- [19] A. B. Montz et al. Scout: A communications-oriented operating system. Technical Report TR 94-20, Department of Computer Science, University of Arizona, June 1994.
- [20] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proc. Globecom'92 IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, Dec. 1992.
- [21] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.
- [22] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148, Amsterdam, 1989. North-Holland.
- [23] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the SIGCOMM '93 Symposium*, pages 64–73, Sept. 1993.
- [24] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.
- [25] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter 1994 Usenix Conference*, pages 153–165, Jan. 1994.

Effects of Buffering Semantics on I/O Performance

José Carlos Brustoloni and Peter Steenkiste

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

jcb@cs.cmu.edu, prs@cs.cmu.edu

Abstract

We present a novel taxonomy that characterizes in a structured way the software and hardware tradeoffs for I/O data passing between applications and operating system. This work contributes new techniques, *input-disabled pageout*, *transient output copy-on-write*, and *input alignment*, that are used for copy avoidance in an optimized buffering semantics, *emulated copy*. Emulated copy offers the same API and integrity guarantees as those of copy semantics and, therefore, can transparently replace it. We implemented an I/O framework, Genie, that allows applications to select any semantics in the taxonomy. Using Genie for communication between PCs and AlphaStations over an ATM network at 155 Mbps, we found that all non-copy semantics performed similarly, and that only copy semantics had distinctly inferior performance. We analyzed end-to-end latencies in terms of the costs of primitive data passing operations and modeled how those costs scale with CPU, memory, and network speeds. The analysis suggests that current trends tend to intensify the observed performance clustering. The main conclusion is that existing I/O interfaces with copy semantics, such as that of Unix, can be transparently converted to emulated copy semantics and thus achieve performance comparable to the best obtainable with any semantics in the taxonomy.

1 Introduction

Most workstation operating systems continue to use I/O buffering schemes with *copy* semantics, similar to that of Unix [15]. In such schemes, the system inputs or outputs data only through system buffers.

This research is sponsored in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0054 and in part by the Advanced Research Projects Agency under contract number F19628-92-C-0116.

On an output call, the system *copies* data from application buffers to system buffers and, conversely, on input completion, the system *copies* data from system buffers to application buffers.

The relative cost of the memory accesses necessary for such copying has increased dramatically since the 1970's, when Unix was introduced. Access times for DRAMs, the almost universal option for workstation main memory, have been improving by roughly only 50% each decade [12]. In contrast, CPU performance has been improving from 50 to 100% per year [12], and local area network (LAN) point-to-point bandwidth, as shown in Table 1, has been increasing by roughly an order of magnitude each decade. Today, LAN bandwidth sometimes actually *exceeds* main memory bandwidth.

LAN	Year introduced	Bandwidth (Mbps)
Token ring	1972	1, 4, or 16
Ethernet	1976	3 or 10
FDDI	1987	100
ATM	1989	155, 622, or 2488
HIPPI	1992	800 or 1600

Table 1: Approximate year of introduction and point-to-point bandwidth of several popular LANs

Data passing between applications and operating system, therefore, has become a major bottleneck in workstation I/O performance [17]. Emerging I/O-intensive applications, such as multimedia, parallel file systems, and supercomputing on clusters of workstations, among others, demand elimination of this bottleneck.

Several previous works have proposed improving data passing efficiency by making I/O operations have *move* [6] or *share* [10, 2] semantics. Move semantics avoids data copying by using typically much cheaper virtual memory (VM) manipulations. On output with move semantics, the system *removes* the region containing the application data from the ap-

plication address space. The application buffer *becomes* the system output buffer, and its pages carry the data without copying, being simply unmapped. Conversely, on input with move semantics, the system inputs data into a system buffer and, after input completion, maps it to a freshly allocated region in the application address space. The system buffer *becomes* the application input buffer and its pages carry the data, again, without copying, being simply mapped. Although efficient, these manipulations also imply that applications cannot access output data after output nor determine the location or layout of input data. This restrictive application programming interface (API) may have prevented move semantics from gaining widespread use.

Share semantics eliminates data copying by performing I/O *in-place*, that is, directly to or from application buffers, without distinct intermediate system buffers. Application buffers are simply *promoted* to double as system buffers during I/O. This promotion may require VM manipulations such as *wiring* the buffer to guarantee that it remains in physical memory. Share semantics can use the same API as copy semantics, but offers lower integrity guarantees on application I/O buffers and may require special hardware support.

We present a novel taxonomy of I/O data passing semantics that extends previous works so as to permit a clear characterization of associated software and hardware tradeoffs. The taxonomy identifies a fourth basic semantics for I/O data passing, *weak move*, and associates with each basic semantics the possible optimizations.

This work contributes two new techniques for safety and correctness of in-place I/O: *I/O-deferred page deallocation* and *input-disabled copy-on-write*. We also introduce four new optimizations: *input-disabled pageout*, *region hiding*, *transient output copy-on-write* (TCOW), and *input alignment*. Input-disabled pageout improves the performance of in-place I/O by making it unnecessary to wire the application buffer during I/O, in the traditional sense of removing its pages from lists where the pageout daemon might find them. Region hiding avoids region allocation and removal costs in the *emulated move* semantics. TCOW and input alignment eliminate data copying in an optimized semantics, *emulated copy*, that offers the same API and integrity guarantees as those of copy semantics and thus can, unlike other semantics in the taxonomy, replace copy semantics without requiring any changes in existing applications.

We implemented a new I/O framework, Genie, that allows applications to select any semantics in

the taxonomy. Using Genie for communication between PCs and AlphaStations over an ATM network at 155 Mbps, we found, rather surprisingly, that all semantics other than copy performed quite similarly. Only copy semantics had distinctly inferior performance.

We analyzed these results in terms of the costs of primitive data passing operations required by each semantics and modeled how those costs scale with CPU, memory, and network speeds. The analysis suggests that, if current trends are maintained, performance differences between semantics other than copy will further decrease, and the performance gap between copy and the other semantics will widen.

The main conclusion of this work is that good I/O performance does not require radical redesign of the I/O API. Existing interfaces with copy semantics can be transparently converted to emulated copy semantics and thus achieve performance that is close to the best that could be obtained with any semantics in the taxonomy.

The rest of this paper is organized as follows. Section 2 presents our taxonomy. Sections 3, 4, and 5 describe techniques for in-place, emulated move, and emulated copy I/O. Section 6 describes the implementation of each data passing semantics in terms of primitive data passing operations. Section 7 shows our experimental results, and Section 8 analyzes them. Section 9 discusses related work, and Section 10 presents our conclusions.

2 Taxonomy of data passing semantics

Data can be passed between applications and operating system according to different semantics. We classify buffering semantics, as shown in Figure 1, in three dimensions: buffer allocation scheme, guaranteed integrity, and level of optimization. The following subsections discuss each dimension in turn.

2.1 Buffer allocation scheme

In *application-allocated* buffering the *application* determines the location of its input buffers on input and retains access to its output buffers after output. This is the case of the *copy* semantics of Unix I/O [15]. In *system-allocated* buffering, on the contrary, the *system* automatically allocates application input buffers on input and deallocates application output buffers on output, and applications cannot or should not access their output buffers after output. System-allocated buffering is typified by the *move* semantics of V [6].

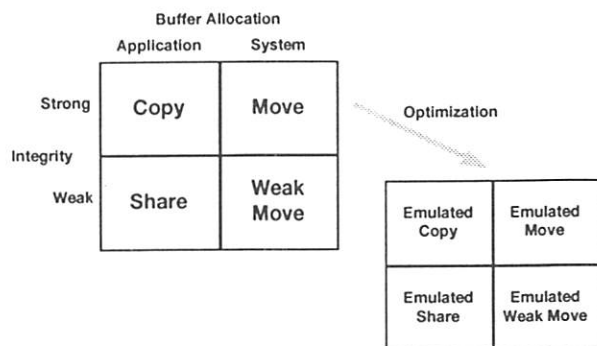


Figure 1: Taxonomy of data passing semantics

Application-allocated and system-allocated buffering require different APIs. The main difference is on input: In the application-allocated API, the application passes the location of its input buffers to the system, while in the system-allocated API, the system returns to the application the location of the newly allocated application input buffers. The system-allocated API also includes calls to allocate or deallocate I/O buffers. Applications with balanced amounts of input and output may be able to avoid explicit buffer allocation and deallocation by using buffers implicitly allocated in input operations for subsequent output operations.

System-allocated I/O buffers can be implemented as regions that are marked *moved in* when accessible by the application. Regions that are not system-allocated are marked *unmovable*. Output with system-allocated semantics is only allowed on *moved in* regions because the resulting deallocation might open inconsistent gaps in *unmovable* regions, such as the heap or the stack.

Compared to application-allocated buffering, system-allocated buffering imposes fewer constraints on the system and thus may be more easily optimized, but whether it results in better end-to-end performance depends on the application. Applications that require access to output buffers after output or that are sensitive to the layout of data, e.g. those using data structures such as arrays, may not be able to use system-allocated semantics without copying between system-allocated I/O buffers and application data structures. This user-level copying may defeat performance advantages of system-allocated semantics.

2.2 Guaranteed integrity

For *strong-integrity* buffering the system guarantees that it will output the data contained in the application output buffer at the time of output in-

vocation, unaffected by subsequent overwriting by the application, and that the application will not observe application input buffers in incomplete or erroneous states during input or after failed input operations. For *weak-integrity* buffering the system makes no such guarantees.

Strong integrity is typified by copy and move semantics, both of which guarantee integrity by physically inputting or outputting data only through system buffers inaccessible by the application.

Weak integrity allows I/O to be performed *in-place*, directly into or from buffers mapped to the application. The application can access these buffers during I/O and, consequently, can corrupt data being output or observe input data in inconsistent states.

Weak-integrity buffering can be application-allocated, which we call *share* semantics, or system-allocated, which we call *weak move* semantics. In weak move semantics, output buffers remain mapped to the application after output, but the application *should not* access them because the system may reuse them for subsequent input and their contents until then are indeterminate. This reuse can be implemented by what we call *region caching*: The system marks the region *weakly moved out* and enqueues it in the corresponding list, per address space, where the system can find it for later reuse. On a subsequent input, the system dequeues the region and marks it again *moved in*. Genie uses region caching for weak move as well as the optimized *emulated weak move* semantics. A similar caching technique is used in the cached volatile fbuf scheme [10].

2.3 Level of optimization

Each of the four basic semantics can be optimized, achieving compatible behavior with normally better performance. Some optimizations may depend on special conditions, and inclusion of this dimension in the taxonomy makes those conditions visible to programmers. Contrary to the other two dimensions, which each had two discrete points, the optimization dimension admits a spectrum of possibilities, including many different from those presented here. We call Genie optimized semantics *emulated* because Genie only uses optimizations that are transparent to applications and do not require changes in programs written for the corresponding basic semantics. The rest of this subsection summarizes previously proposed optimizations, and the following three sections describe novel optimizations used in Genie.

Output with copy semantics can be performed in-place if the region containing the output data is

made *copy-on-write* (COW) [18]. The system removes write permissions from the application's mapping of the pages in the region. Attempts by the application to overwrite a page in this region will cause a VM fault. The system recovers from this fault by copying the data in the page to a new page and mapping this new page to the same virtual address in the application address space, with writing enabled. The application cannot overwrite output pages, which guarantees integrity, and copying only occurs if the application does attempt overwriting. A simpler alternative to COW, reported to have better performance [1], is to remove output page write permissions and mark output pages busy during I/O. Attempts by the application to overwrite an output page then cause the application to fault and stall until output completes.

Input with copy semantics can be optimized, if the application buffer is page-aligned and of length multiple of a page size, by *swapping* pages between system and application buffers. Swapping unmaps application buffer pages and then maps system buffer pages to the same virtual addresses. This optimization is present in SGI and HP systems.

I/O with share and weak move semantics can be optimized by requiring application I/O buffers to be located in special non-pageable areas [2, 3], which eliminates the need to wire the buffers during I/O.

3 Safe and efficient in-place I/O

Previous proposals have often restricted in-place I/O to special regions, such as *exposed buffer areas* [2] or *fbuf regions* [10]. In this section we describe how Genie makes in-place I/O safe and efficient regardless of data location.

3.1 I/O-deferred page deallocation

Wiring is sufficient to guarantee that a system buffer will remain in physical memory during I/O because only the pageout daemon might asynchronously deallocate system buffer pages. In the case of application buffers, however, wiring is insufficient because other events may also cause application memory deallocation. These events include normal or abnormal termination of the application and explicit memory deallocation by the (possibly malicious) application. If deallocated pages still have pending I/O when they are reused for a different process, there may be corruption of output data or of the other process's memory.

Genie makes in-place I/O safe by keeping counts of input and output references to each physical page

in current input and output operations. Genie integrates in what we call *page referencing* the activities of preparing the descriptor with the physical addresses of an I/O request, verifying access rights, and updating reference counts.

Genie changes the system page deallocation routine to refrain from placing pages with nonzero input or output count in the list of free pages, whence they might be allocated to other processes. After completing an I/O operation, Genie *unreferences* pages by updating their reference counts. If a given page no longer has any input or output references, Genie verifies whether the page is still allocated to a memory object; if not, Genie assumes that the page was deallocated during I/O, and enqueues it in the list of free pages for reutilization. We call this scheme *I/O-deferred page deallocation* and use it for all in-place I/O.

3.2 Input-disabled pageout

Genie modifies the pageout daemon to refrain from paging out pages with nonzero input reference count. Pending input would modify these pages *after* pageout, making paged out data inconsistent. Moreover, the application that invoked the input is likely to access these pages after input, making them poor candidates for pageout. Genie allows the daemon to page out pages with zero input count normally, regardless of output reference count.

This optimization, *input-disabled pageout*, adds no overhead to page referencing and makes wiring unnecessary in Genie's emulated semantics, both on input and on output. Note that input-disabled pageout allows application data to be arbitrarily located and does not reduce the memory available to the rest of the system, unlike previously proposed optimizations requiring special non-pageable buffer areas [2, 3].

3.3 Input-disabled COW

COW is frequently used to optimize interprocess communication or memory inheritance with copy semantics [18]. However, it may not implement copy semantics correctly if there is a pending in-place input operation in the region. Indeed, if the input is by DMA, the input will modify memory without generating any write faults, even though the pages are mapped read-only to the applications. Consequently, changes may be observed by processes other than the one that issued the input, and COW in this case actually implements share rather than copy semantics.

Genie guarantees correctness in this case by also monitoring the total number of input references to pages of each memory object in current input operations. Updating this count is integrated with page referencing and unreferencing. Genie modifies the system COW routine to perform a physical copy rather than COW if any of the region's backing memory objects has nonzero input count.

The reverse case, when a region is COW before in-place input, does not require special handling, because input page referencing verifies write access rights, which will automatically fault-in a private, writable copy of the data.

4 Move emulation with region hiding

Genie uses *region hiding* to implement a new semantics, *emulated move*, that is compatible with move semantics but performs I/O in-place. On output with emulated move semantics, Genie removes read and write permissions to pages in the output region, marks the region *moved out*, and enqueues it for later reuse, as in region caching. Genie modifies the system VM fault routine to recover from faults only in *unmovable* or *moved in* regions. Attempts by the application to access the region after output will therefore cause unrecoverable VM faults, as would be the case if the region had actually been removed, but the region and its pages remain allocated to the application address space. On a subsequent input Genie reuses the region, marks it *moved in*, and reinstates page read and write permissions.

5 Optimizations for copy emulation

5.1 TCOW

Conventional COW can be too expensive for output copy avoidance [1]. Page referencing, however, allows Genie to implement a specialized, page-level form of COW, TCOW, that is highly efficient for this purpose.

On output with emulated copy semantics, Genie simply references and removes write permissions to the application output pages. Attempts by the application to overwrite output pages will cause VM faults. Genie modifies VM fault processing as follows, in the case of write faults in regions for which the application has write permissions, when the page is found in the top memory object backing the region [18]: If the output count of the page is nonzero, the system recovers from the fault by copying the contents of the page to a new page, swapping pages in the memory object, and mapping the new page

to the application buffer with writing enabled; if the output count is zero by the time the write fault occurs, the system recovers by simply reenabling writing on the page (no copying). If the page is found but not in the top object, the fault is a conventional COW fault and is handled normally. Note that TCOW adds to page referencing only the cost of removing page write permissions, which is arguably the minimum necessary overhead for strong-integrity, safe in-place output.

TCOW differs from conventional COW in two ways. First, TCOW is transient — COW is conventionally long-term, while TCOW only lasts during output, which is when it is actually useful. Second, TCOW operates at page level instead of at region level. This allows TCOW to prevent the proliferation of regions on output and reduce the number of VM data structures that it needs to manipulate. TCOW performs as well as the busy-marking scheme [1, 3] with the added benefit of not stalling applications that overwrite output buffers during output.

5.2 Input alignment

On input with emulated copy semantics, Genie inputs data into system buffers that start at the same page offsets and have the same lengths as the corresponding application buffers. Consequently, Genie can swap pages between system and application buffers even if application buffers are not page-aligned or have lengths that are not multiple of the page size. This scheme, *system input alignment*, goes against the traditional practice of allocating system buffers without regard to the alignment and length of application buffers. As illustrated in Figure 2, lack of alignment makes it impossible to swap pages.

For situations where the system is unable to align its buffers, Genie offers *application input alignment*, where the *application* aligns its buffers to system buffers. Genie includes an interface through which applications can query I/O modules (e.g. a protocol stack) about the preferred alignment and length of input buffers. The preferred alignment may be nonzero and the preferred length may be not equal to a multiple of the page size because, for example, of unstripped packet headers and network maximum transmission units.

Genie uses what we call *reverse copyout* to pass data in partially filled pages in aligned buffers. If data in the system page is shorter than the reverse copyout threshold, Genie simply copies it out, as shown for item 1 in Figure 2. If it is longer, however, Genie first completes the rest of the system

	Prepare	Dispose
Copy	Allocate system buffer. Copyin output data.	Deallocate system buffer.
Emulated copy	Reference application pages. Read-only application pages.	Unreference application pages.
Share	Reference application pages. Wire region.	Unwire region. Unreference application pages.
Emulated share	Reference application pages.	Unreference application pages.
Move	Reference application pages. Wire region. Mark region <i>moving out</i> . Invalidate application pages.	Unwire region. Unreference application pages. Remove region.
Emulated move	Reference application pages. Mark region <i>moving out</i> . Invalidate application pages.	Unreference application pages. Mark region <i>moved out</i> and enqueue.
Weak move	Reference application pages. Wire region. Mark region <i>moving out</i> .	Unwire region. Unreference application pages. Mark region <i>weakly moved out</i> and enqueue.
Emulated weak move	Reference application pages. Mark region <i>moving out</i> .	Unreference application pages. Mark region <i>weakly moved out</i> and enqueue.

Table 2: Operations for data output from application to kernel

page with corresponding data from the respective application page and then swaps the pages, as shown for items 3 and 4 in Figure 2. The threshold is set just above half the page size so as to minimize data copying.

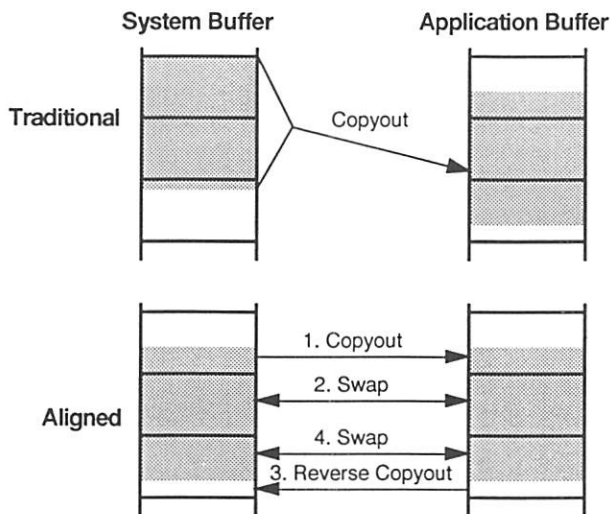


Figure 2: Input alignment

6 Data passing implementation in Genie

This section describes, in terms of primitive data passing operations, how Genie implements each semantics for data passing between user-level applications and I/O modules (such as protocol stacks and drivers) in the kernel. The following subsections discuss output and input in turn. We will use these detailed descriptions to analyze experimental

results in the next two sections.

Both on output and on input, Genie takes advantage of the fact that copy semantics often is very efficient for short data. If data is shorter than configurable thresholds, Genie automatically converts emulated copy or emulated share semantics into copy semantics.

6.1 Output

Output data passing involves two processing stages: *prepare*, when the application invokes the output operation, and *dispose*, when output completes and the system is ready to return control to the application. The operations used for output are summarized in Table 2, where “read-only” means “remove write permissions”, and “invalidate” means “remove all access permissions” in the corresponding VM page table entries.

Note that Genie does not remove a system-allocated region until dispose time in order to guarantee that the corresponding virtual addresses will not be reassigned during I/O, thus allowing graceful recovery in case of error.

6.2 Input

Input data passing involves three processing stages: *prepare*, when the application invokes the input operation; *ready*, when the device needs buffering; and *dispose*, when the input operation is complete and the system is ready to return control to the application.

Input processing must match the type of input buffering used by the device controller. Genie distin-

	Prepare	Ready	Dispose
Copy		Allocate system buffer.	Copyout input data. Deallocate system buffer.
Emulated copy		Allocate aligned buffer.	Swap pages. Deallocate aligned buffer.
Share	Reference application pages. Wire region.		Unwire region. Unreference application pages.
Emulated share	Reference application pages.		Unreference application pages.
Move		Allocate system buffer.	Create region. Zero-complete system pages and fill region. Map region and mark <i>moved in</i> .
Emulated move	Dequeue <i>moved out</i> region, mark region <i>moving in</i> , and reference application pages.		Check region, unreference application pages, reinstate page accesses, and mark region <i>moved in</i> .
Weak move	Dequeue <i>weakly moved out</i> region, mark region <i>moving in</i> , and reference application pages. Wire region.		Check region. Unwire region. Unreference application pages and mark region <i>moved in</i> .
Emulated weak move	Dequeue <i>weakly moved out</i> region, mark region <i>moving in</i> , and reference application pages.		Check region, unreference application pages, and mark region <i>moved in</i> .

Table 3: Operations for data input with early demultiplexing

guishes three types of device input buffering: *early demultiplexed*, *pooled in-host*, and *outboard*, as described in the following.

6.2.1 Input with early demultiplexing

With early demultiplexed input buffering, buffers reside in host main memory and are specified by multiple (pointer, length) pairs. The device controller keeps separate input buffer lists per input request or connection and inputs data directly into a buffer from the appropriate list.

Early demultiplexing enables in-place or system-aligned buffering *if* the application informs the system about the location of its buffers *before* physical input. This condition is trivially met when physical input is synchronous to application requests. In data communication, however, input may occur before solicited by the application, and location information must be provided either by the receiver, using a pre-posted, possibly asynchronous input operation, or by the sender, using a field in the packet header [5, 20] or implicitly by the connection used [16]. If location information is not available to the system, copy avoidance is still possible by application-aligned or system-allocated buffering.

Table 3 summarizes the operations used for pre-posted input with early demultiplexing.

Note that, to maintain protection, move semantics has to clear the unused portions of a system buffer before mapping it to the application. If the semantics is emulated move, weak move, or emulated

weak move and at prepare time no suitable cached region can be found in the appropriate queue, Genie allocates a new region and marks it *moving in*. For the same three semantics, Genie checks, at dispose time, that the cached region prepared and used for input is still present in the application address space. If it was removed (advertently or not) by the application, Genie maps the corresponding pages to a new region, guaranteeing that the location information returned to the application correctly points to the input data.

6.2.2 Input with pooled buffering

With pooled in-host buffering, the device controller allocates input buffers from a pool of fixed-size buffers (commonly pages) in host main memory without considering the corresponding input request or connection. This is still the most popular of the input buffering schemes, although early demultiplexing is becoming more common with the advent of ATM networks.

Pooled buffering does not allow in-place or system-aligned buffering, and copy avoidance is possible only with application-aligned or system-allocated buffering.

Genie always prepares application input buffers according to Table 3, enabling early demultiplexing. Actual input may use pooled buffering, however, either because the device does not support early demultiplexing or because the application did not inform the location of its input buffers before physi-

	Ready	Dispose
Copy	Allocate overlay buffer. Overlay buffer.	Copyout input data. Deallocate overlay buffer.
Emulated copy	Allocate overlay buffer. Overlay buffer.	If aligned, swap pages, else copy out. Deallocate overlay buffer.
Share	Allocate overlay buffer. Overlay buffer.	Unwire region. Unreference application pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Emulated share	Allocate overlay buffer. Overlay buffer.	Unreference application pages. If aligned, swap pages, else copy out. Deallocate overlay buffer.
Move	Allocate overlay buffer. Overlay buffer.	Create region. Zero-complete overlay pages, fill region and refill overlay buffer. Map region and mark <i>moved in</i> . Deallocate overlay buffer.
Emulated move Emulated weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unreference application pages. Swap pages. Mark region <i>moved in</i> . Deallocate overlay buffer.
Weak move	Allocate overlay buffer. Overlay buffer.	Check region. Unwire region. Unreference application pages. Swap pages. Mark region <i>moved in</i> . Deallocate overlay buffer.

Table 4: Ready and dispose-time operations for input with pooled buffering

cal input. For pooled buffering, the ready-time and dispose-time operations are those shown in Table 4. At ready time the I/O module inputs data into *overlay buffers* allocated from a private pool of pages in main memory. At dispose time, Genie passes data from overlay buffers to application buffers and deallocates the overlay buffers, returning them to the respective I/O module pool for reuse.

Note that in the case of move semantics, Genie maps overlay pages to the application and therefore needs to refill the overlay buffer with the same number of newly-allocated pages to avoid pool depletion.

6.2.3 Input with outboard buffering

With outboard buffering, the device controller allocates input buffers from a pool in outboard memory. Outboard buffers can be transferred directly into application buffers after successful input completion, providing strong integrity regardless of the semantics of application buffers and even if the system was not previously informed of the location of application buffers.

However, outboard buffering can also add complexity and cost to the controller. In the context of network adapters, early demultiplexed and pooled in-host buffering are examples of “cut-through” architectures, while outboard buffering has a “store-and-forward” architecture that typically imposes higher latency.

If the device uses outboard buffers for input, Genie alters the operations in Table 3 as follows: For all semantics other than emulated copy, at ready time, after the operations in the table, start DMA into host memory; and at dispose time, after the operations in the table, deallocate the outboard buffer.

For emulated copy semantics, no buffer is allocated at ready time, and at dispose time, the system references the application pages, DMA's data from the outboard buffer to the application buffer, unreferences the application pages, and deallocates the outboard buffer. Consequently, with outboard buffering, emulated copy is implemented much as emulated share semantics.

7 Experimental comparison of data passing semantics

This section reports end-to-end latencies and CPU utilization for datagram communication measured using the various buffering semantics and early demultiplexed or pooled input buffering. We ran our experiments on computers of the types shown in Table 5, connected by the Credit Net ATM network [14] at OC-3 (155 Mbps) rates. Unless otherwise noted all figures and tables in this and the next sections refer to the Micron P166 PCs. Results for the other platforms were similar and can be found in [3]. The Credit Net network adapter transfers data between main memory and the physical medium by burst-mode DMA over the PCI I/O bus. We used the NetBSD 1.1 operating system augmented with an implementation of the Genie interface, through which applications accessed the network. We measured latencies by capturing the value of the CPU on-chip cycle counter at appropriate points in the code. All measurements were made on warm caches and are the averages of five runs. We report primarily latencies rather than throughput to simplify analysis.

	Micron P166	Gateway P5-90	DEC AlphaStation 255/33
CPU	Pentium 166 MHz	Pentium 90 MHz	21064A 233 MHz
Integer rating	4.52	< 2.88	< 3.48
L1-cache	8 KBI + 8 KBD, 3560 Mbps	8 KBI + 8 KBD, 1910 Mbps	16 KBI + 16 KBD, 2860 Mbps
L2-cache	256 KB, 486 Mbps	256 KB, 244 Mbps	1 MB, 1366 Mbps
Memory	32 MB, 4 KB page, 351 Mbps	32 MB, 4 KB page, 146 Mbps	64 MB, 8 KB page, 350 Mbps

Table 5: Characteristics of the computers used in the experiments. The integer rating used for the Micron P166 is the listed SPECint95 of the Dell XPS 166. The rating taken as upper bound for the Gateway P5-90 is the listed SPECint95 of the Dell XPS 90, which has a bigger and faster L2-cache. The rating taken as upper bound for the AlphaStation is its listed SPECint.base95 because the version of NetBSD used on it could not be compiled with optimizations. The cache and memory bandwidths listed are the peak values we observed using a *bcopy* benchmark at user level.

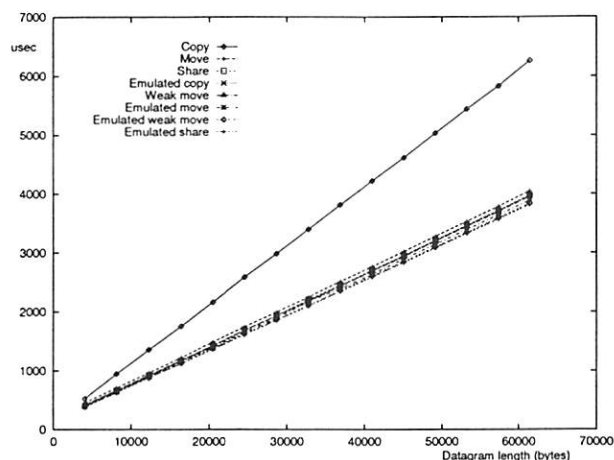


Figure 3: End-to-end latency with early demultiplexing. Semantics other than copy have similar performances.

Figure 3¹ shows latencies using early demultiplexing. We measured latencies for increasing datagram lengths equal to a multiple of the page size, up to 60 KB, the largest such multiple allowed by ATM AAL5. For these datagrams, copy semantics gave much higher latency than did any of the other semantics, which pass data using VM manipulations instead of copying. The differences between semantics other than copy were small. The most striking difference was that between copy and emulated copy semantics, which offer the same API and integrity guarantees. Using TCOW and input alignment, emulated copy reduced latencies for 60 KB datagrams by 37%. For all data lengths emulated copy also gave lower latency than those of move and share semantics. This advantage is due to the use of input-disabled pageout instead of region wiring in

¹In all figures in this section, we list the semantics in the legend in the same order as the respective curves. For fine discrimination between curves that appear cluttered in the figure, please refer to Table 7.

emulated copy semantics. Emulated move semantics gave slightly lower latencies than those of emulated copy semantics because it simply invalidates and re-instates page table entries instead of fully swapping pages, which also requires updating the respective memory object. Latency was still lower, but only slightly, with the emulated weak-integrity semantics, which do not require page table updates. The equivalent throughput for single 60 KB datagrams was 78 Mbps for copy, 121 Mbps for move, 124 Mbps for share, emulated copy, and weak move, 126 Mbps for emulated move, 128 Mbps for emulated weak move, and 129 Mbps for emulated share semantics.

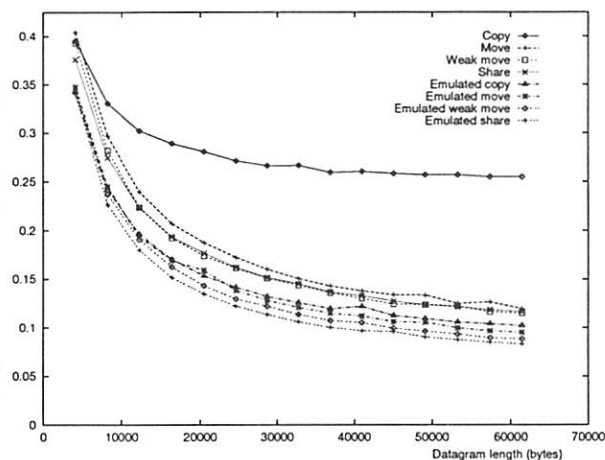


Figure 4: CPU utilization. Copy semantics leaves much less CPU time available for applications.

We instrumented the idle loop in the operating system scheduler to measure CPU idle time while performing the experiment of Figure 3. We show the results in Figure 4. CPU utilization was much higher for copy semantics than for any other semantics. For 60 KB datagrams, the utilizations were 26% for copy, 12% for move, weak move, and share, 10% for emulated copy and emulated move, 9% for emulated weak move, and 8% for emulated share se-

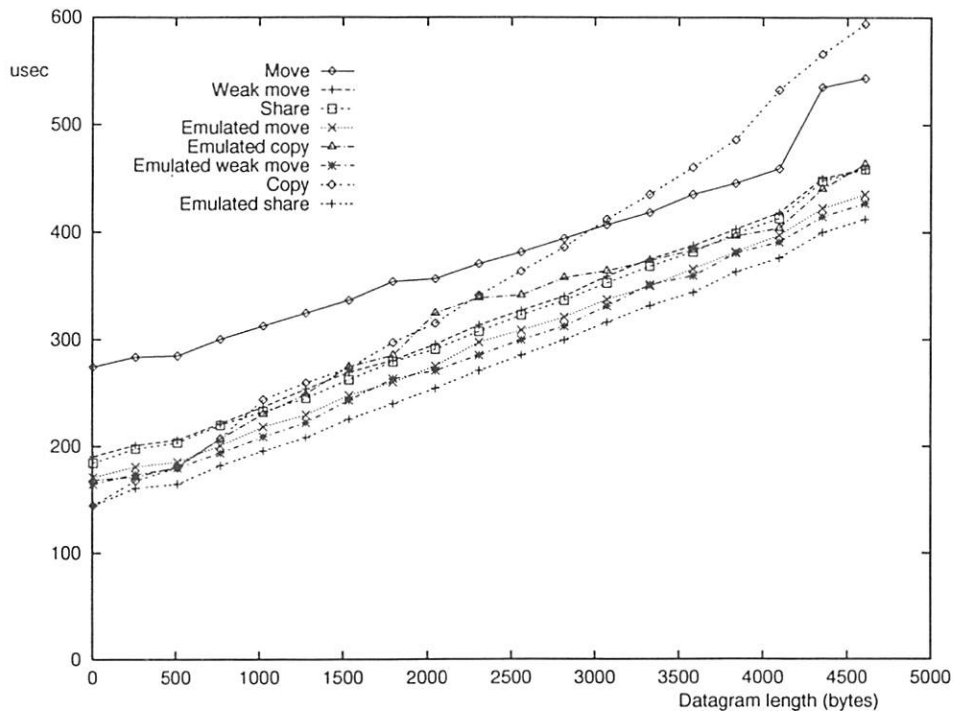


Figure 5: End-to-end latency for short datagrams with early demultiplexing. Using reverse copyout, emulated copy avoids copying more than about half a page.

mantics.

Figure 5 shows short datagram latencies with early demultiplexing. Move semantics gave by far the highest latency for short datagrams because it has to zero the part of the page not occupied by data. Emulated move semantics gave much lower latencies because it performs I/O in place, using region hiding, and therefore does not need to zero the remainder of the page. Copy semantics gave the lowest (145 μ sec) but also the most rapidly rising latency because of the high incremental cost of copying.

We set output thresholds so that Genie automatically converted to copy semantics output of data shorter than 1666 bytes with emulated copy or 280 bytes with emulated share semantics. We set the reverse copyout threshold at 2178 bytes. (Performance is only moderately sensitive to these settings; we empirically determined these values to give good results.) With these settings, emulated copy semantics had about the same latency as that of copy semantics for data up to half page long; above that, reverse copyout and swapping significantly reduced the latency of emulated copy relative to that of copy semantics. Emulated share had, for all data lengths, the lowest latency, because its data passing overhead consists solely of page referencing. The difference between latencies with emulated copy and emulated share semantics was maximal at half page size: 325

vs. 254 μ sec. Weak move and emulated weak move semantics gave slightly higher latencies than those of share and emulated share, respectively, because of region caching costs avoided in application-allocated semantics. The slightly higher latency of emulated move semantics relative to that of emulated weak move semantics is due to region hiding. The higher latencies of share and weak move semantics relative to their emulated counterparts are due to region wiring and unwiring, which cost about 35 μ sec for the first page and become unnecessary in the emulated semantics because of the input-disabled page-out optimization.

Figure 6 shows latencies with pooled input buffering and application-aligned application buffers. Copy and emulated copy have latencies only very slightly higher than the respective latencies with early demultiplexing, corresponding to the same operations plus buffer overlay overhead. Share, move, and weak move semantics have higher latencies than those of emulated copy because of region wiring and unwiring. All other semantics have latencies very close to that of emulated copy. For single 60 KB datagrams, the equivalent throughput is 77 Mbps for copy, 120 Mbps for share, move, and weak move, 123 Mbps for emulated move, emulated copy and emulated weak move, and 124 Mbps for emulated share semantics.

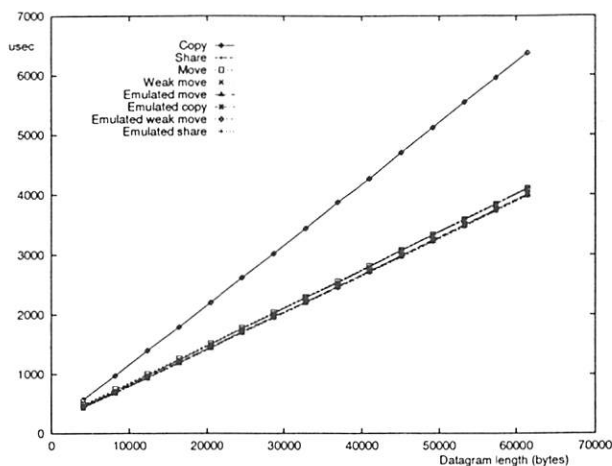


Figure 6: End-to-end latency with application-aligned pooled input buffering. If there is alignment, non-copy application-allocated semantics give performances similar to those of system-allocated semantics.

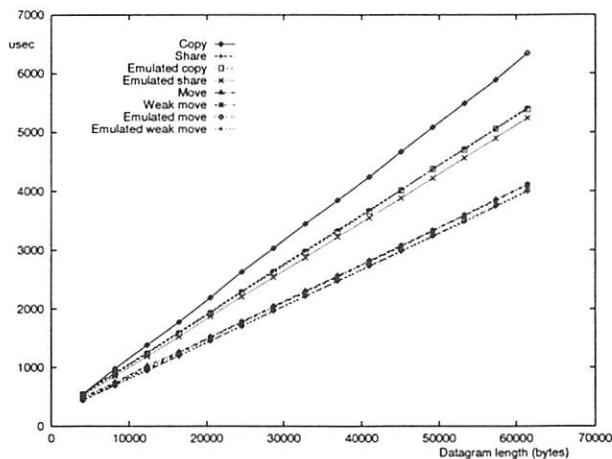


Figure 7: End-to-end latency with unaligned pooled input buffering. Without alignment, application-allocated semantics require copying at the receiver.

Figure 7 shows latencies with pooled input buffering and unaligned application buffers. In this case, emulated copy, share, and emulated share semantics require data copying on input, whereas the other semantics are unaffected. This figure clearly shows the impact of data copying, splitting the semantics into a group with no copies (system-allocated semantics), another with two copies (copy semantics, with one copy at the sender and another at the receiver), and the remaining group, between the other two, with one copy. For single 60 KB datagrams, the equivalent throughput is 77 Mbps for copy semantics, around 92 Mbps for the other application-allocated semantics, and 121 Mbps for the system-

allocated semantics.

Figure 7 may give the impression that system-allocated semantics are intrinsically more efficient than application-allocated semantics if pooled buffering is used. However, if an application is insensitive to data layout enough to use system-allocated semantics, then in principle that application can also align its buffers to system buffers, and then emulated copy, emulated share, and system-allocated semantics give very similar performance, as shown in Figure 6. If, on the contrary, an application is sensitive to data layout, it would require application-level copies between system-allocated I/O buffers and application data structures. The total number of copies (possibly one copy on output, if the application needs to retain access to the same or other data on the same pages, plus one copy on input) is then at best the same as if emulated copy or emulated share semantics were used (one copy on input only). In those cases, system-allocated semantics may actually give worse end-to-end performance than application-allocated semantics.

We do not show results with outboard buffering because of limitations in the hardware used. However, we expect that, compared to early demultiplexing, the staging of data at an outboard buffer will add an equal amount of latency to all semantics except emulated copy. Because of the special way the latter is handled, we expect it to have performance even closer to that of emulated share semantics.

8 Analysis

This section analyzes empirical end-to-end latencies in terms of the costs of primitive data passing operations and models how those costs scale with CPU, memory, and network speeds. We use this model to discuss the sensitivity of our results.

End-to-end latencies can be *broken down* into the sum of a *base* latency and *data passing* latencies at the sender and receiver. The base latency captures end-to-end costs that are independent of the particular buffering semantics or input buffering scheme used, such as crossing the application-kernel boundary and incurring driver, device, network, and interrupt latencies. Data passing latencies, on the contrary, depend on the semantics and input buffering scheme used. We take the base latency to be equal to the end-to-end latency of emulated share semantics with early demultiplexing, reduced by the costs of referencing and unreferencing application I/O buffers.

Only *prepare* time data passing operations at the sender contribute to end-to-end latency, because

Operation	Latency	Operation	Latency
Base	$0.0598 B + 130$	Swap	$0.00163 B + 15$
Copyin	$0.0180 B - 3$	Copyout	$0.0220 B + 15$
Reference	$0.000363 B + 5$	Unreference	$0.000100 B + 2$
Wire	$0.00141 B + 18$	Unwire	$0.000237 B + 10$
Read only	$0.000367 B + 2$	Region create	24
Invalidate	$0.000373 B + 2$	Region fill	$0.000398 B + 9$
Region mark out	3	Region fill & overlay refill	$0.000716 B + 11$
Overlay allocate	7	Region map	$0.000474 B + 6$
Overlay	7	Region check, unreference, reinstate, mark in	$0.000507 B + 11$
Overlay deallocate	$0.000344 B + 12$	Region check, unreference, mark in	$0.000194 B + 6$
Region check	5		
Region mark in	1		

Table 6: Costs of primitive data passing operations, in μsec . B is the data length in bytes.

Semantics		Early demultiplexing	Application-aligned pooled	Unaligned pooled
Copy	E	$0.0997 B + 141$	$0.100 B + 166$	$0.100 B + 166$
	A	$0.0998 B + 125$	$0.101 B + 139$	$0.101 B + 144$
Emulated copy	E	$0.0621 B + 153$	$0.0625 B + 178$	$0.0828 B + 177$
	A	$0.0622 B + 150$	$0.0622 B + 175$	$0.0848 B + 195$
Share	E	$0.0619 B + 165$	$0.0637 B + 204$	$0.0841 B + 203$
	A	$0.0621 B + 162$	$0.0638 B + 197$	$0.0846 B + 219$
Emulated share	E	$0.0602 B + 137$	$0.0621 B + 175$	$0.0825 B + 175$
	A	$0.0600 B + 137$	$0.0619 B + 167$	$0.0824 B + 178$
Move	E	$0.0628 B + 197$	$0.0634 B + 224$	$0.0634 B + 224$
	A	$0.0626 B + 202$	$0.0631 B + 234$	$0.0631 B + 234$
Emulated move	E	$0.0610 B + 151$	$0.0625 B + 185$	$0.0625 B + 185$
	A	$0.0609 B + 150$	$0.0623 B + 183$	$0.0623 B + 183$
Weak move	E	$0.0620 B + 173$	$0.0637 B + 212$	$0.0637 B + 212$
	A	$0.0615 B + 170$	$0.0633 B + 206$	$0.0633 B + 206$
Emulated weak move	E	$0.0603 B + 144$	$0.0621 B + 183$	$0.0621 B + 183$
	A	$0.0602 B + 143$	$0.0619 B + 184$	$0.0619 B + 184$

Table 7: Estimated (E) and actual (A) end-to-end latencies, in μsec . B is the data length in bytes.

dispose-time operations overlap with network latencies and latencies at the receiver. Conversely, with early demultiplexing, prepare and ready time operations at the receiver overlap with latencies at the sender and in the network, and only the *dispose* time operations at the receiver contribute to end-to-end latency. With pooled or outboard buffering, only *ready* time and *dispose* time operations at the receiver contribute to end-to-end latency.

We directly measured the latencies of primitive data passing operations by instrumenting the Genie code. We added instructions at appropriate points in the code to record the value of the CPU on-chip cycle counter. We recorded the time intervals for each operation and datagram length when performing the experiments reported in Figures 3, 6, and 7, taking the averages of five runs. We then performed a least-squares linear fit on each operation latency versus datagram length. We obtained excellent correlation except in cases of constant or very

small latencies. We averaged the fitted equations of each operation latency over the semantics and input buffering schemes where the operation is used. We show the results in Table 6. Note that *copyin* cost less than *copyout* because our experiments were on warm caches. On output (*copyin*), data can be read from the cache, while on input (*copyout*) it has to be read from memory. The *copyin* cost is actually nonlinear because the L1-cache has much higher bandwidth than that of the L2-cache. This causes a negative y-intercept in the corresponding linear fit.

Taking the values from Table 6, we added, for each semantics, the base latency, the costs of the respective output prepare-time operations indicated in Table 2, and the costs of the respective input dispose-time operations indicated in Table 3, obtaining an estimate of the respective end-to-end latency with early demultiplexing. We show these estimates in Table 7, along with the least-squares linear fit of the actual end-to-end latencies from Figure 3.

Type of Parameter	Gateway P5-90				AlphaStation 255/233			
	Estimated	GM	Min	Max	Estimated	GM	Min	Max
Memory-dominated	2.40	2.43	2.43	2.43	1.00	0.83	0.83	0.83
Cache-dominated	> 1.44, < 3.33	2.46	2.46	2.46	> 0.26, < 1.39	0.54	0.54	0.54
CPU-dominated multiplicative factor	> 1.57	1.79	1.58	1.92	> 1.30	1.64	0.75	3.77
CPU-dominated fixed term	> 1.57	1.83	1.53	2.59	> 1.30	1.54	0.47	3.74

Table 8: Scaling of data passing costs relative to the Micron P166. “GM”, “Min”, and “Max” are the geometric mean, minimum, and maximum values of the ratios of parameters of each given type.

Adding base latency, the costs of output prepare-time operations, and the costs of input ready-time and dispose-time operations indicated in Table 4, for each semantics, we obtained estimates of the respective end-to-end latencies with pooled buffering and application-aligned or unaligned application buffers. We show these estimates in Table 7, along with the least squares linear fit of the actual end-to-end latencies from Figures 6 and 7. The good fit between estimated and actual latencies suggests that our breakdown model is accurate for the datagram lengths considered, which are multiples of the page size (additional terms would increase accuracy for intermediate lengths, but also make the model more complicated).

Using the breakdown model, the end-to-end latency when sender and receiver use different semantics can be expected to be equal to the sum of the base latency plus sender-side latencies of the semantics used by the sender plus receiver-side latencies of the semantics used by the receiver.

The breakdown model can be extended into a *scaling* model that takes into account CPU, memory, and network speeds. To a first approximation: (1) The multiplicative factor of the base latency is *network-dominated* and equal to the inverse of the net network transmission rate, subject to adapter and I/O bus bandwidth limitations; (2) The fixed term of the base latency is equal to the sum of I/O bus, device, and network latencies, plus a term corresponding to fixed operating system overhead, which scales inversely to CPU speed; (3) The *copyout* multiplicative factor is *memory-dominated* and equal to the inverse of the main memory copy bandwidth, and the associated fixed term can be ignored; (4) The *copyin* multiplicative factor is *cache-dominated* and may vary between the inverse of the copy bandwidth of the L2-cache and the inverse of the copy bandwidth of main memory, depending on data and cache sizes and cache associativity and locality. The fixed term can be ignored; (5) All other parameters are *CPU-dominated* and scale according to CPU speed, as estimated by an appropriate integer bench-

mark, such as SPECint95².

We verified (1), (3), and (4) in each platform [3]. Table 8 shows the verification of (3), (4), and (5) across platforms. Agreement between estimated and actual scaling was quite good for the Gateway P5-90, which has the same architecture as the base case. In the AlphaStation, CPU-dominated ratios had geometric means consistent with the model but variances that were much higher than those of the Gateway P5-90. This could be expected, given that the AlphaStation has a substantially different architecture.

Extrapolating based on the scaling model, if CPU speeds continue to increase faster than transmission rates, as is the current trend, the performance differences between semantics other than copy will tend to decrease, and if CPU speeds continue to increase faster than main memory bandwidth, the performance difference between copy and other semantics will increase. At OC-12 (622 Mbps) rates, the scaling model predicts that the end-to-end throughput for single 60 KB datagrams with early demultiplexing on the Micron P166 PCs will be close to 140 Mbps with copy semantics, 404 Mbps with emulated copy semantics, 463 Mbps with emulated share semantics, or 380 Mbps with move semantics, giving emulated copy almost three times better performance than that of copy semantics.

9 Related work

Previous work in I/O buffering has typically focused on optimizations using a particular semantics, device, or application. We are not aware of other direct, controlled comparisons covering as broad a range of buffering options as we present here.

The technique discussed here for emulated copy input with outboard buffering is a generalization of

²The cost of page table updates may scale otherwise between processors of different architecture, causing the cost of the *read-only*, *invalidate*, *swap*, *region map*, and *reinstall* operations to diverge from this model. Page table updates are particularly costly in multiprocessors, where the semantics that do not use these operations – weak (with early demultiplexing) and copy – may have some advantage.

that used in [19, 9]. However, those works also use outboard buffering for copy elimination on output. We, on the contrary, avoid copying on output by simple VM manipulations, which may simplify device interface design. Staging output through an outboard buffer may still be advantageous, however, if, for example, there is hardware to compute the TCP checksum (which goes in the packet header) while data is being DMAed from application buffer to outboard buffer (which was the case in [19]).

Fbufs [10] are system-allocated but are optimized with mixed semantics. Cached fbuf output has semantics similar to emulated copy, but requires wiring and leaves buffers read-only until explicit deallocation because there is no COW scheme. Cached volatile fbuf output has semantics similar to share. Cached and cached volatile fbuf input have semantics similar to weak move but use read-only buffers that must be deallocated explicitly.

We assumed the use of DMA for I/O, which is necessary for high throughput in most current systems. Programmed I/O may make it unnecessary to wire application buffers, reducing latency, but subject to page faults, which could cause deadlock if the faulted process is itself a (possibly user-level) pager or is invoked by a pager. In our implementation, input-disabled pageout eliminates wiring costs safely and cheaply. Programmed I/O may have the advantage of simplifying the implementation of early demultiplexing.

There have been proposals to reduce the penalty of copying by integrating it with other data touching operations, such as TCP checksumming [7]. Integration of checksumming on input has semantic implications: If checksumming is integrated with the copy from device or system buffer to application buffer, and the checksum is wrong, the application buffer will be overwritten with faulty data, and the semantics is actually weak and not copy. If a system buffer is involved (i.e., not programmed I/O between device and application buffers), in our implementation, at least for long data, it costs less to pass the data by VM manipulation and then read it for checksumming than to read and write (one-step checksum and copy) the data [4].

The work reported here concerns I/O data path (buffering) optimizations, but we would like to point out recent work on I/O control-path optimizations, such as bypassing the operating system (OS) [8, 11, 21] and separating control from data transfer [22, 16]. In terms of the analysis from the previous section, bypassing the OS reduces the constant term of the base latency (i.e., the latency for short data). Given that the wiring of application buffers requires

OS intervention, OS bypassing may imply some form of either non-pageable buffer areas or application-level programmed I/O between application buffers and device controller.

10 Conclusions

We introduced a new taxonomy that characterizes in a structured way a very broad and general range of options for I/O data passing between applications and operating systems. We described the implementation of data passing in Genie, a new I/O framework that allows applications to select any data passing semantics in the taxonomy. Using an implementation of Genie on NetBSD, we made direct, controlled evaluations and comparisons of the different buffering semantics. We identified the fundamental similarities and differences between the various semantics, assessed the impact of different architectural support in device controllers, and investigated how performance scales with CPU and memory speeds.

The experiments demonstrated significant performance gains due to TCOW, input alignment, region hiding, and input-disabled pageout. The results indicate that large I/O performance improvements are possible in many Unix-derived operating systems by using emulated copy instead of copy semantics. This substitution does not require changes in applications because both semantics give the same integrity guarantees and can offer the same API.

At the I/O rates tested, some additional performance improvement is possible by using emulated share semantics, particularly at short data lengths. The scaling model predicts that larger relative improvements would occur when interfacing with faster devices or in shared-memory multiprocessors. Although emulated share semantics can offer the same API as copy semantics, the lower integrity guarantees may require changes in applications. We expect that few or no changes would be necessary in multiprogrammed or distributed applications that already lock and checkpoint data.

Contrary to what might have been expected, we did not find system-allocated semantics to have any consistent advantage relative to application-allocated semantics. In fact, we found that application-aligned, application-allocated buffers differ very little in functionality or performance from system-allocated buffers. Because system-allocated semantics may require either substantial rework of pre-existing applications written for copy semantics or application-level copies that negate performance improvements, we believe that emulated copy and em-

ulated share semantics offer more practical and general approaches for I/O performance improvement.

Acknowledgements

Prashant Chandra and Todd Mummert helped with experimental set-up. Anonymous referees and the OSDI paper "shepherd", Willy Zaenepoel, provided many helpful suggestions.

References

- [1] J. Barrera III. "A fast Mach network IPC implementation", in *Proc. USENIX Mach Symp.*, USENIX, Nov. 1991, pp. 1-11.
- [2] J. Brustoloni. "Exposed Buffering and Sub-Datagram Flow Control for ATM LANs", in *Proc. 19th Conf. Local Computer Networks*, IEEE, Oct. 1994, pp. 324-334.
- [3] J. Brustoloni and P. Steenkiste. "Application-Allocated I/O Buffering with System-Allocated Performance". Tech. Rep. CMU-CS-96-169, Carnegie Mellon Univ., Pittsburgh, PA, 1996.
- [4] J. Brustoloni and P. Steenkiste. "Copy Emulation in Checksummed, Multiple-Packet Communication". Submitted for publication.
- [5] G. Buzzard, D. Jacobson, M. Mackey, S. Marovitch and J. Wilkes. "An implementation of the Hamlyn sender-managed interface architecture", in *Proc. OSDI'96*, USENIX, Oct. 1996.
- [6] D. Cheriton. "The V distributed system", in *Comm. ACM*, 31(3):314-333, March 1988.
- [7] D. Clark and D. Tennenhouse. "Architectural considerations for a new generation of protocols", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 200-208.
- [8] E. Cooper, P. Steenkiste, R. Sansom and B. Zill. "Protocol Implementation on the Nectar Communication Processor", in *Proc. SIGCOMM'90*, ACM, Sept. 1990, pp. 135-143.
- [9] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards and J. Lumley. "Afterburner", in *IEEE Network*, July 1993, pp. 36-43.
- [10] P. Druschel and L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 189-202.
- [11] P. Druschel, L. Peterson and B. Davie. "Experience with a High-Speed Network Adaptor: A Software Perspective", in *Proc. SIGCOMM'94*, ACM, Aug. 1994, pp. 2-13.
- [12] J. Henessy and D. Patterson. "Computer Architecture: A Quantitative Approach". Morgan Kaufmann Pub., San Mateo, CA, 1990.
- [13] K. Kleinpaste, P. Steenkiste and B. Zill. "Software Support for Outboard Buffering and Checksumming", in *Proc. SIGCOMM'95*, ACM, Aug. 1995, pp. 87-98.
- [14] C. Kosak, D. Eckhardt, T. Mummert, P. Steenkiste and A. Fischer. "Buffer Management and Flow Control in the Credit Net ATM Host Interface", in *Proc. 20th Conf. Local Computer Networks*, IEEE, Oct. 1995, pp. 370-378.
- [15] S. Leffler, M. McKusick, M. Karels and J. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Pub. Co., Reading, MA, 1989.
- [16] T. Mummert, C. Kosak, P. Steenkiste and A. Fischer. "Fine Grain Parallel Communication on General Purpose LANs", in *Proc. 10th Intl. Conf. Supercomputing*, ACM, 1996, pp. 341-349.
- [17] J. Ousterhout. "Why aren't operating systems getting faster as fast as hardware?", in *Proc. Summer 1990 Conf.*, USENIX, June 1990, pp. 247-256.
- [18] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", in *Proc. 2nd ASPLOS*, ACM, Oct. 1987, pp. 31-39.
- [19] P. Steenkiste, B. Zill, H. Kung, S. Schlick, J. Hughes, R. Kowalski and J. Mullaney. "A Host Interface Architecture for High-Speed Networks", in *Proc. 4th IFIP Conf. High Performance Networks*, IFIP, Dec. 1992, pp. A3 1-16.
- [20] T. Stricker, J. Stichnoth, D. O'Hallaron and S. Hinrichs. "Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers", in *Proc. Intl. Conf. Supercomputing*, ACM, July 1995, pp. 1-10.
- [21] T. von Eicken, A. Basu, V. Buch and W. Vogels. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", in *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 40-53.
- [22] C. Thekkath, H. Levy and E. Lazowska. "Separating Data and Control Transfer in Distributed Operating Systems", in *Proc. 6th ASPLOS*, ACM, Oct. 1994, pp. 2-11.

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX technical symposia and system administrator conferences are the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:

- problem-solving with a practical bias,
- fostering innovation and research that works,
- communicating rapidly the results of both research and innovation,
- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login:*, the Association's bi-monthly newsletter featuring technical articles, system administration tips and techniques, an international calendar of events, SAGE News, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web <<http://www.usenix.org>>.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and mobile computing – as many as seven technical meetings every year.
- Discounts on the purchase of proceedings from USENIX conferences and symposia and other technical publications.
- Discount on BSDI, Inc. products. BSDI information: 800 800 4BSD.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. (800 998 9938) and USENIX.
- Discount on all publications and software from Prime Time Freeware, including Prime Time Freeware for Unix, Prime Time Freeware for AI, Prime Time TeXcetera and Tools & Toys for UnixWare. Contact <ptf@ptf.com>.
- Savings (10-20%) on selected titles from McGraw-Hill (212 512 2000), The MIT Press (800 356 0343), Prentice Hall (201 592 2657), John Wiley & Sons (212 850 6789), and O'Reilly & Associates (800 998 9938) and Uniforum.
- Special subscription rates to the periodicals *The Linux Journal* (206 527 3385), *UniForum Monthly*, *UniNews*, and the annual *UniForum Open Systems Products Directory* (800 255 5620).
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

ANDATACO
Apunix Computer Services
Crosswind Technologies
Earthlink Network, Inc.
Frame Technology Corporation
ISG Technologies, Inc.
Matsushita Electrical Industrial Co., Ltd.
Motorola Research & Development

Open Market, Inc.
Shiva Corporation
Sybase, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

SAGE Supporting Members:

Bluestone, Inc.
Enterprise Systems Management Corporation
Great Circle Associates

Pencom Systems Inc.
Southwestern Bell
Taos Mountain

For further information about membership, conferences or publications, contact: The USENIX Association,
2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: +1-510-528-8649 Fax: +1-510-548-5738
Email: office@usenix.org URL: <http://www.usenix.org>

ISBN 1-880446-82-0